SPEを256倍使い倒すための本

完結編

nosuke著

SPEを256倍使い倒すための本 完結編 nosuke著

電脳のツボ

SPEを256倍使い倒すための本 完結編

nosuke著

── 表紙・裏表紙の生き物 ──

90nm さん

初代 PS3 の中の人．TDP 100W オーバーの熱血さんですが，たまに熱すぎて暴走します．生まれはテキサス州オースチン．趣味は食べることと 65nm さんを愛でること．

65nm さん

2 代目 PS3 の中の人．たぶん今一番出回っている子．日本は長崎県生まれ．90nm さんに比べて TDP も胸囲もだいぶ小さくなりました orz．アツい 90nm さんに振り回される毎日ですが，なぜか楽しそう．趣味は物理演算とたんぱく質計算．

45nm さん

3 代目 PS3 の中の人．本体もちっちゃくなっている新人さんです．クールになりすぎてデンパさんになってしまいました．日々 (法定内で) EMI を放射中．政治的な理由で Linux が動かない唯一の子．でも頑張り屋さんです．

おーすちん

Cell さんたちの心の故郷．上の階では N 天堂，下の階では Micro$oft のプロセッサと作っていたという噂も．

[おうちのかたへ]

- 本書に掲載されているプログラムのソースやその断片は，2009 年 12 月現在，以下の筆者の Web ページよりダウンロードすることができます．また，本書の内容の補足・訂正なども，以下の Web ページより取得することができます．

  電脳のツボ サークルページ
  `http://garakuta.homelinux.org/circle/`

- 本書の内容についてはなるべく正確を期することを心がけていますが，残念ながらその内容を保証するものではありません．本書を趣味や研究に活用して頂くことや，教材などとしてご利用頂くことについては一切の制限は設けませんが，万一本書に掲載された情報に基づいて何らかの損害が発生したとしても，筆者は一切の責任を負いかねます．あらかじめご了承ください．

- 本書に記載されている会社名や製品名，商品名などは，一般に各社の登録商標または商標です．本書では ™，©，®などのマークは省略しています．

- 本書に掲載されている Hack the Cell 2009，メルセンヌ・ツイスタおよび newlib 由来のソースコード片のライセンスは，いずれも引用元のソースコードのライセンスに従います．また，それ以外の筆者オリジナルのソースコード片については，修正 BSD ライセンスに従うものとします．

- 本書の一部または全部を著作権法の定める範囲を超えて無断で複写，複製，転載などを行うことは，お互い恥ずかしいのでご遠慮ください．

はしがき

初めての方ははじめまして！そうでない方はこんにちは！電脳のツボの nosuke です．本書をお手に取ってくださりありがとうございます．

本書は PS3+Linux+Cell SDK という環境における，C 言語での SPE プログラミングについて解説を行った技術系同人誌になります．説明するまでもないかもしれませんが，SPE は PS3 の CPU である Cell/B.E. に内蔵されている演算専門のプロセッサコアです．

いきなりネガティブな話をすると，SPE は一般的なプロセッサコアと比べて様々な点が異なるため，C 言語でのプログラミングに慣れたプログラマでもなかなか使いこなすことができないという弱点を抱えています．たとえば SPE が直接参照できるメモリは 256KByte しかないため，大きなデータを処理するには演算処理だけでなく，外部とデータを効率的にやりとりする処理もあわせてプログラミングしなければならないとか，ハードウェアがその場で実行コードを最適化してくれないので，何も知らずに C 言語の入門書に書いてあることを信じてプログラミングをしてしまうと理論ピーク性能の 1 割も性能が出ないとか，きちんとしたアプリケーションを作ろうと思うと，SPE のプログラムを並列化した上で，さらにそれを制御する汎用プロセッサコア (PPE) 用のプログラムを別途作らなければならないとか … とにかくそれなりのものを作り上げるまでに存在する挫折ポイントが少なくありません．実際に Hello World よりも先の世界に踏み込んだ経験があるならば，その手間のかかり具合とご機嫌取りの難しさを身をもって味わったことでしょう．筆者も幾度となく味わいました．

しかしながら，その一方で，SPE 自体は元々非常に高い演算能力を備えたプロセッサコアであるため，SPE に適した処理を，SPE 向けにきちんとチューニングしてあげることで，最新のパソコン用の CPU を遥かに上回る処理能力を実現することができます．先ほど述べたような挫折ポイントに潜むアーキテクチャやコンパイラなどの特徴やクセについて調べ，考え，試行錯誤を繰り返した結果，それまで亀のような速度でしか動かなかった処理が数十倍もの速度で動くようになったときの爽快感や達成感もまた，実際に SPE プログラミングに本気で取り組んだ人にしかわからないものです．“この感覚に惹かれて SPE プログラミングにハマった” なんていう人も少なからずいるのではないでしょうか．筆者もそんな 1 人です．

筆者としては，是非とも多くの人にこの爽快感や達成感を味わって欲しいと思うわけですが，ここに 1 つ問題があります．この Cell/B.E. の SPE の魅力を簡潔に伝えるのにちょうど適した “何か” がないのです．友人にお気に入りのミュージシャンの音楽の良さを伝えたいときは，そのミュージシャンのイチオシのアルバムを貸せばよく，旅で訪れた土地の素晴らしさを伝えたければ，旅先で撮った写真とお土産に少々の話を添えれば十分でしょう．でも，SPE プログラミングには，そういうものがないのです．SCEI や IBM の Web サイトに行けば，膨大で詳細で汎用的なドキュメントを得ることができますが，よほどのマニアでもない限り，そこから SPE

プログラミングの魅力を感じ取ることは難しいでしょう．また，有志によっていくつかの情報がWeb 上にまとめられていたりもしますが，その多くは Linux をインストールしてみたという話やちょっとしたプログラムをコンパイルして動かしてみた的な体験談が中心で，SPE プログラミングの魅力を伝えるまでのものはほとんどありません．

これは非常にもどかしい話です．こんなに面白いものがあるのに，その魅力をうまく伝えるのに適した “何か” がないのです．だから，仕方がないので，自分で作っちゃいました．その “何か” を．ファンが自分が好きなものを他の人にも知って欲しいので勝手に作ったファンブック的技術書．それが本書になります．

本書の概要に関しては第 1 章にてあらためて紹介しますが，“ここから始めて，これを理解して，そしてこういう風にプログラミングをすれば SPE の能力を引き出すことができる” ということをまとめた本書は，“Cell/B.E. や SPE のプログラミングに興味があるけどどこから手をつければよいのかわからない” という方にとって恰好のガイドとなることでしょう．また，既に SPE のプログラミングを経験しているものの，いまいちコツがつかめなくて困っている方や，あるいは筆者と同じように誰かと SPE プログラミングの魅力を共有したいけれど，気軽に紹介できる手頃な参考書がないとお困りの方にも本書は役に立つのではないかと思います．本書を読んで，SPE プログラミングに新たに挑戦してくれる方や，何かを再発見してくれる方，Cell/B.E. 界隈を盛り上げてくれるような方が 1 人でも現れれば幸いです．

ところで，実は，本書は完全な書き下ろしではありません．本書は，2009 年 8 月のコミックマーケット 76 (C76) において頒布した “SPE を 256 倍使うための本 – 力の 16 倍編 –” と “SPE を 256 倍使うための本 – 技の 16 倍編 –” の内容を 1 冊にまとめ，さらに大幅な加筆修正を行ったものになります．

前作では時間やページ数などの都合で書きたくても書けなかったことが大量にあったのですが，今回はそれらを可能な限り盛り込みました．特に，プログラムの最適化に関しては，前作ではよい題材が見つからなかったこともあって，概要のみの紹介にとどまり，具体例を示すことができなかったわけですが，今回は実際にベンチマーク風のプログラムを順を追って最適化していく過程を書き足し，最適化の流れを数字とソースコードの両方から実感できるようになりました．また，同様に，前作ではほとんど触れることのなかったプログラムの並列化に関しても，新規に章を 1 つ書き起こし，プログラムの並列化に必要となる基礎的な知識などとともに，具体的なプログラムの並列化の流れを紹介するようにしました．それから，PS3 への Linux や Cell SDK のインストールといった作業の手順についても，簡単ではありますが新たに記載をしました．おかげでより入門書らしくなったのではないかと自負しております．もちろん，一方では stop-and-signal 命令を用いた外部関数呼び出しの実装と使い方の紹介や，MFC のプロブレムステート領域のレジスタに SPE からアクセスして DMA 転送を行う方法の紹介なども新たに書き起こし，“誰得” な内容もより一層充実させました．加筆分だけでも C76 版の 1 冊分以上のボリュームとなっています．

C76 版をお買い上げ頂いた方も，そうでない方も，ともに楽しめる内容としたつもりですので，是非最後までお読み頂ければと思います．

目次

はしがき i

第 1 章 はじめに 1
1.1 この本は何？ 1
1.2 この本を読み進めるには何が必要？ 3
1.3 本書の流れ 6
1.4 本書での表記 7

第 2 章 Cell/B.E. と SPE のアーキテクチャ 9
2.1 アーキテクチャとインプリメンテーション 9
2.2 Cell/B.E. の全体構成 9
2.3 PPE 10
2.4 SPE 11
2.5 アドレス空間とメモリモデル 14
2.6 実際の Cell/B.E. 16

第 3 章 開発環境の準備 17
3.1 色々な開発環境の構成 17
3.2 開発環境の構築に必要となるもの 19
3.3 PS3 への Fedora 9 のインストール 20
3.4 PS3 への Cell SDK のインストール 24

第 4 章 SPE を 1 個だけ使うプログラム 27
4.1 SPE のプログラムを呼び出す流れ 27
4.2 SPE のプログラムを呼び出す API 28
4.3 Hello World 30

第 5 章 SPE の演算機能 35
5.1 SPE の SIMD 演算 35
5.2 バイトの並びとビットの番号 36
5.3 ベクタとスカラ 38
5.4 組み込み関数による SPE の演算処理の利用 42
5.5 演算処理向けの型総称組み込み関数 45
5.6 SPU のパイプライン 72
5.7 SPE の浮動小数点数 73
5.8 SIMD 数学ライブラリ 77

第 6 章 SPE のプログラムの最適化 79
6.1 SPE のプログラムの最適化とは 79
6.2 C 言語プログラマのためのアセンブリ言語超入門 80
6.3 SIMD 化 82
6.4 分岐の最適化 95
6.5 ループの最適化 104
6.6 SPU のパイプラインのチューニング 118
6.7 データの参照の最適化 123
6.8 関数のインライン展開 125
6.9 デクリメンタで実行時間を測る 127

第 7 章 Linux と SPE 129
7.1 spufs 129
7.2 libspe2 132

第 8 章 SPE を複数使うプログラム 135
8.1 複数コンテキストの同時実行 135
8.2 複数の SPE を同時に動かす例 136

第 9 章 チャネルとメッセージング機構 141
9.1 チャネル 141
9.2 メールボックス 144
9.3 シグナル 152

第 10 章 DMA 転送 155
10.1 DMA 転送 155
10.2 フェンスとバリア 165
10.3 リスト DMA 169
10.4 アトミック更新 173
10.5 LS 間の DMA 転送 182
10.6 PPE からの DMA 転送の利用 184
10.7 DMA 転送の最適化 188

第 11 章 プログラムの並列化とデータ転送のチューニング 191
11.1 SPE プログラムの並列化 191
11.2 並列化後のプログラムのチューニング 202

第 12 章 プロブレムステート領域 209
12.1 プロブレムステート領域のレジスタのアドレス 209
12.2 DMA 転送要求・完了確認 210
12.3 メールボックスの利用 215
12.4 シグナルの利用 216
12.5 SPE から他の SPE の MFC に DMA 転送を要求する例 217

第 13 章 外部関数呼び出し 223
13.1 stop-and-signal と外部関数呼び出し 223
13.2 標準で提供されている外部関数呼び出し 224
13.3 独自の外部関数の追加 226
13.4 コールバック関数での引数の参照と戻り値の設定 229
13.5 コールバック関数の追加と呼び出しの具体例 230
13.6 `__send_to_ppe` の内部処理と stop-and-signal 命令の生成 233

第 14 章 イベント 235
14.1 イベントの概要と管理 235
14.2 イベントの種類 236
14.3 イベントの検出と処理 237

あとがき 243

索引 245

参考文献 249

第 1 章
はじめに

正義なき力が無力であるのと同時に 力なき正義もまた無力なのですよ
— 三条陸・稲田浩司『ダイの大冒険』より —

1.1 この本は何？

本書は Cell Broadband Engine (Cell/B.E.) における Synergistic Processor Element (SPE) プログラミングのちょっとマニアックな入門書です．

1.1.1 Cell/B.E. と SPE

Cell/B.E. は，ソニー・コンピュータエンタテインメント (SCEI)，ソニー，東芝，IBM によって共同開発された，Cell Broadband Engine Architecture (CBEA) と呼ばれるアーキテクチャ規格に準拠したマルチコアプロセッサの一種です．Cell/B.E. やその他の CBEA に準拠したプロセッサは，Intel の Core i7 や AMD の Phenom のような流行りの対称型のマルチコアプロセッサ (対等な関係にある同じコアのみで構成されたマルチコアプロセッサ) と異なり，PowerPC Processor Element (PPE) と呼ばれる PowerPC 命令互換の汎用的な制御用のコアと，8 個の SPE と呼ばれる独自アーキテクチャの演算用のコアで構成される，非対称型の構成をとっています．

Cell/B.E. は，演算処理に特化したコアを 8 個備えたことで，プロセッサ全体を 4.0GHz で動作させた場合に，SPE だけで最大 256GFLOPS もの演算能力を発揮することができ，発表当初から大きな注目を集めてきました[*1]．アーキテクチャや計測方法が異なるために単純な比較はできませんが，3.2GHz 動作の Core i7 965 の FLOPS 値が 51.2GFLOPS [*2] であることから考えると，Cell/B.E. が世に出てから 3 年以上過ぎた今日においても，数値の上では依然として侮ることのできない高い演算能力を備えたプロセッサであるということが言えるでしょう[*3]．

[*1] 実際に市場に出回っている Cell/B.E. は 3.2GHz 動作で 7 コアまでしか有効になっていないため，約 180GFLOPS が理論ピーク性能となります．

[*2] `http://www.intel.co.jp/jp/support/processors/sb/CS-023143.htm`

[*3] あくまでも数字の上での話です．実際の演算能力は，アプリケーションの処理内容やプロセッサの周辺まで含めたハードウェア構成，その他の様々な要因によって大きく左右されます．Cell/B.E. は強い個性を持ったユニークなプロセッサであることは間違いないと思いますが，数字が示すほど他のプロセッサに対してあらゆる分野で優位かというと，そういうわけではないということは，今日の Cell/B.E. の流行具合を見れば察しがつくことでしょう．

1.1.2 Cell/B.E. を搭載した製品たち

Cell/B.E. は，2006 年 11 月に発売された SCEI の家庭用ゲーム機である PlayStation3 (PS3) に CPU として搭載されており，その演算処理の中核を担う SPE は，ゲームにおける 3 次元画像処理や物理演算，AI 処理などはもとより，DVD 再生時のアップスケーリングなどのマルチメディア系の処理や，タンパク質の構造解析などの科学演算などにも活用されています．また，Cell/B.E. は，東芝のハイエンドデジタルテレビである CELL REGZA にも CPU として採用されており，ソフトウェアによる動画の高画質化や複数チャンネルの同時デコードといった，従来のデジタルテレビとは一線を画した機能の実現にも一役買っています．

一方で，IBM は，Cell/B.E. を元に PowerXCell 8i と呼ばれるプロセッサを開発し，これを搭載した QS22 と呼ばれるブレードサーバを販売しています．PowerXCell 8i は，Cell/B.E. と同じ CBEA 準拠のプロセッサですが，SPE の演算能力が大幅に強化されており，科学演算などにより利用しやすくなっています．この QS22 を搭載したスーパーコンピュータである Roadrunner は，世界のスーパーコンピュータのランキングである Top500 において，初登場の 2008 年 6 月発表のランキングから 2009 年 6 月発表のランキングまで 1 位の座に君臨していました．2009 年 11 月発表のランキングでは，残念ながら首位の座を Cray のコンピュータに明け渡して 2 位となってしまいましたが，それでも相対的に高い演算能力を実現していることに変わりはなく，純粋な演算処理においても CBEA に準拠したプロセッサが高い能力を発揮できることが示されています．

また，東芝は，Cell/B.E. の派生品として，H.264/MPEG2 ハードウェアコーデックに Cell/B.E. と同等の SPE を 4 個組み合わせた SpursEngine と呼ばれるプロセッサを開発し，販売しています．SpursEngine は，東芝の AV ノート PC である Qosmio シリーズに搭載されており，ジェスチャー認識や動画のアップスケーリングなどに活用されています．また，Leadtek やトムソン・カノープスからは，SpursEngine を搭載した動画処理用拡張カードなどが販売されています．SpursEngine は Windows と Linux の両方に対応しており，開発環境も無償で提供されているため，汎用性や拡張性に優れた Intel 系の PC と組み合わせることで初めて実現可能となるような，新たな SPE のアプリケーションの登場にも期待が持てそうです*4.

1.1.3 あらためてこの本について

話が少々それましたが，本書はこの Cell/B.E. の SPE をターゲットにした，C 言語によるプログラミングについて扱います．

SPE は独自の命令セットを備えたコアですが，当然ながら専用のコンパイラが用意されており，標準的な C 言語でプログラムを記述して，それをコンパイルし，生成されたバイナリ (実行コード) を SPE 上でネイティブに動かすことが可能です．C 言語を習いたての人でも，手順さえ踏めば簡単に Hello World を動かすことができるでしょう．

しかしながら，単に C 言語ができるだけではその能力を十分に引き出すことができないのが SPE プログラミングの難しいところです．SPE は演算に特化したコアであるがゆえに，近年の

*4 筆者もカードは買ってあるので是非何か作ってみたいところではあるのですが ….

一般的な PC 向けのプロセッサや組み込み向けのプロセッサとは大きく異なる特徴や制約を多数備えています．そのため，SPE 本来の能力を存分に引き出すには，SPE ならではの特徴を活かし，SPE であるがゆえの制約をうまく回避した実行コードが必要となります．そして，それを実現するには，C 言語でプログラミングを行う段階で様々な事柄に気を遣わなければならないのです．

本書では，このような SPE 特有の "気を遣うべき事柄" に関して，筆者が知りうる範囲の情報を一通り紹介しています．その際，"こういうときはこうすべきである" だけで済ませずに，あわせて "なぜそうなっていて，どうしてこうするとよいのか" ということまで記述しています．したがって，本書の内容は，SPE の性能を引き出すためのコツを知りたいという方だけでなく，アーキテクチャやコンパイラのことまで配慮した，より深いプログラミングを経験したいという方の役にも立つことでしょう．

また，その一方で，本書では他であまり取り上げられることのない SPE のマニアックな機能や使い方などについても色々と紹介しています．各種ドキュメントに "できる" とだけ書かれているものの，具体的にどのようにすれば利用できるのかがはっきりと示されていない機能を中心に，実用性度外視で取り上げています．もちろん，これらについても "こうすれば使えます" と紹介して終わりにせずに，まずその仕組みがハードウェアや周辺のソフトウェアによってどのように実現されているのかを示した上で，実際の使い方を紹介しています．SPE や Cell/B.E. を搭載したシステム全体で，従来と違った新しいことをやってみたいと考えている方には，もしかしたら何かのヒントになるかもしれません．

1.2　この本を読み進めるには何が必要？

1.2.1　予備知識

本書を読み進めるにあたっては，まずそれなりの予備知識が必要となります．本書は以下の前提知識がある読者を想定して書かれています．

- C 言語に関する基本的な知識と認識

  資格テストではありませんので，関数ポインタの書き方や標準関数の引数の順番，細かな演算子の優先順位などを正確に暗記している必要はありませんが*5，ポインタの概念や記述方法が理解できていないと厳しいでしょう．また，"C 言語で記述したプログラムは，そのまま抽象的な形で処理されるのではなく，最終的にプロセッサが処理できる命令の並びに変換されて実行される" ということをきちんと認識できていないと，やはり厳しいかもしれません．

- プログラミングに関するそこそこの経験と知識

  本書は SPE プログラミングの入門書ではありますが，プログラミングそのものの入門書ではありません．そのため，本書は読者にある程度のプログラミング経験があるものとして

*5 そんなものは忘れたら man を参照して確認すればよいだけのことです．

書かれています．プログラミング経験がまったくない場合，本書の内容についていくのはかなり大変ではないかと思われます．

また，本書の後半では，スレッドを利用したプログラムが登場しますが，これに関しても概念程度は知っているものとして書かれています．スレッドを生成するための API の引数を丸暗記しているかどうかといったことはまったく重要ではありませんが[*6]，スレッドとは何かといったことがわからないと，やはり苦しいです．

一方で，MPI や OpenMP などを利用した並列プログラミングの経験は特に必要ありません．もちろん並列プログラミングに慣れていた方が，そうでない場合と比べて理解が早い場面は多々あるとは思いますが，本書で扱う SPE プログラミングでは，非常に低いレベルでハードウェアを直接叩くことでプロセッサ間の通信を行うため，高度に抽象化された既存の並列プログラミングのフレームワークにおける経験がそのまま役に立つようなシチュエーションは少ないものと考えられます．

- UNIX/Linux 系 OS の基本的な操作方法

  本書では，PS3 に Linux をインストールして，その上で SPE プログラムを動かします．もちろん，Windows 上でお好みのエディタを使ってコーディングを行うことはできますが，その場合でもコンパイルや実行は Linux 上での作業となるため，UNIX/Linux 系 OS の操作に関する基礎的な知識がないと，色々と戸惑うことになるかと思います．UNIX/Linux 系 OS が苦手な人や嫌いな人は，これを機に勉強してみるとよいかもしれません．

- コンピュータアーキテクチャや OS に関する基礎知識

  本書は SPE のプログラミングについて扱った本ではありますが，SPE の特徴や制約などを紹介するにあたり，コンピュータアーキテクチャや OS に関する用語を特に説明なく使うことが多々あります．これらに関しても，ある程度の基礎知識が必要となるでしょう．もっとも，大抵のことはネットで調べればわかる時代ですので，わからない単語が出てきたら Google や Wikipedia の力を借りて勉強しながら読み進めるのもよいかもしれません．

1.2.2　開発環境

本書を手を動かしながら読み進めたい場合は，Cell/B.E. の SPE 向けの開発環境が必要となります．もちろん，手を動すことなく本書を単なる読み物や置物として楽しむというのであれば開発環境は不要ですが，プログラミングが好きな方であれば，第 5 章あたりから実際にプログラムを動かしてみたくて仕方なくなるのではないかと思われます．

Cell/B.E. の SPE 上でネイティブ動作するソフトウェアを開発する場合，開発に必要となるツールと，それを利用する環境，それからツールを利用して生成したソフトウェアを実行することのできる Cell/B.E. を搭載したハードウェアが必要となります[*7]．このようなツールやハードウェアにはいくつか種類がありますが，個人で手軽に利用できるものとなると限られます．

[*6] そんなものは忘れたら m(ry

[*7] ソフトウェアを実行するだけならシミュレータを利用することもできますが，あまり現実的ではないことから，ここでは考えないことにします．

開発ツール

個人で手軽に利用できる Cell/B.E. 向けの開発ツールには，IBM などによって提供されている Cell SDK があります．Cell SDK は，Cell/B.E. の PPE 上で動作する一般的な PowerPC 向けのソフトウェアと，SPE 向けのソフトウェアの両方の開発に対応した SDK で，GCC などの GNU ツールチェインをベースとしたオープンソース系のコンパイラやデバッガなどの他，spu_timing や SIMD 数学ライブラリといった補助的なツールやライブラリなどを含んでいます．

開発ツール利用環境

Cell SDK をインストールして利用することができるのは，Intel 系アーキテクチャのマシン (32bit・64bit) および PowerPC を搭載したマシンに限られており，対応している OS も Linux のみとなっています．したがって，Cell SDK のツールを利用して開発を行うには，Fedora 9 などの Cell SDK が対応している Linux ディストリビューションのインストールされた Intel 系の CPU を搭載した PC (もしくは PowerPC を搭載した PC など) が必要となります．手近に空いている PC があれば，Web から Fedora 9 の ISO イメージをダウンロードしてインストールするだけで Cell SDK を利用可能な環境が整います．また，Linux を直接インストール可能な手頃な PC がない場合や，Fedora 9 を PC に直接インストールすることに抵抗がある場合などでも，既存の OS 上で VMware Player などの仮想マシンを動かし，その上に Fedora 9 などの Cell SDK がサポートする Linux ディストリビューションをインストールすることで，開発ツールの利用が可能となります．いずれにしても，それなりの性能の PC が 1 台あれば，お金をかけることなく，Cell SDK を利用した開発が可能となります．

実行環境

一方，開発したソフトウェアを実行するのに必要となる Cell/B.E. を搭載したハードウェアについてですが，1.1.2 で述べたように Cell/B.E. やその派生品を搭載した機器には様々なものが存在するものの，個人で手軽に入手でき，なおかつ自由に自作のソフトウェアを実行できる環境となると，2009 年 12 月の時点では旧型の PS3 [*8]一択となります[*9]．PS3 に PowerPC 向けの Linux ディストリビューションをインストールし，ゲーム OS と呼ばれる SCEI の提供するゲーム向けの OS の代わりに Linux を起動しておくことで，個人で開発したプログラムを PS3 上で実行することが可能となります．

なお，Cell SDK が PowerPC 向けの Linux 上で利用可能であり，さらに PS3 自体に PowerPC 向けの Linux がインストール可能である (というよりもインストールしないと Cell SDK で開発した自作のプログラムを実行することができない) ことから，当然ながら PS3 のみを利用して Cell/B.E. 向けのソフトウェア開発を行うことも可能です[*10]．

[*8] 2009 年 9 月に発売された新型の PS3 では，Linux などの他の OS の起動ができなくなってしまったため，実際に自分で開発したソフトウェアを実行する環境として利用可能な PS3 は旧型のものに限られます．

[*9] SpursEngine でも SPE 向けプログラムの実行が可能ですが，本書では主として Cell/B.E. における SPE を用いたプログラミングを扱うため，ここでは SpursEngine は対象外としています．

[*10] ただし PS3 は搭載するメモリが 256MByte と少なく，PPE が遅いせいかコンパイラなどの動作も Intel 系の PC 上で動かした場合と比べて大変緩慢なため，快適な環境で開発を行いたいのであれば，PS3 単体で開発を行うことはおすすめしません．

1.2.3　参考資料

本書は，巻末の参考文献のところに記載されているドキュメントや書籍を一次ソースとして利用して書かれています．本書を読んでいて疑問に思った点やもっと深く知りたいと思った事柄がある場合は，これらを当たってみてください．PDF のものはどれも無償で公開されており，自由にダウンロードすることができます[*11]．

また，本書では，きちんとした形でドキュメント化されていない情報についても扱っています．この辺りの情報は，主に libspe2 と newlib (SPE 用の libc) のソースコードを読んで取得していますので，必要であればこれらを入手して参考とするのもよいでしょう．libspe2 と newlib のソースコードは，Barcelona SuperComputing Center (BSC) のサーバ[*12] よりダウンロードすることができます．

1.3　本書の流れ

本書では，次章以後で，SPE プログラミングに必要となるアーキテクチャの知識から実際のコーディング例まで幅広く紹介していきます．

第 2 章では，SPE を中心とした Cell/B.E. 全体のハードウェアアーキテクチャについて紹介します．ここを押さえておかないと SPE プログラミングは始まりません．ハードウェアと関係の深い用語についてもここでいくつか紹介していますので，Cell/B.E. や SPE のことを Web の記事などを読んである程度知っている方も一読することをおすすめします．

第 3 章では，SPE プログラミングを行う上で必要となる開発環境の構築手順について簡単に紹介します．既に Cell SDK を利用できる環境があり，PS3 に Linux がインストールしてある方は読み飛ばしていただいて結構です．

第 4 章では，実際に C 言語でプログラムを記述してコンパイルし，SPE 上で動かすまでの手順を示します．動かすプログラムは単なる Hello World なのですが，巷の C 言語の教科書に書かれている Hello World と同じと思って読み飛ばすと，後々ついてこられなくなるので，SPE プログラミングの経験がない方は是非一度目を通しておくことをおすすめします．

第 5 章では，SPE プログラミングにおいて性能を引き出す上で重要な要素の 1 つである SIMD 演算を中心に，SPE の演算処理に関する話題を扱います．この章では，前半で SPE の演算における重要事項を一通り紹介し，後半で SPE の持つすべての演算機能について図を交えて解説を行います．この章は非常にボリュームがあり，特に後半は同じような話題が延々と続くため，いきなりきっちり読み切るのはとても大変です．まずは前半をしっかり読んで理解した上で，後半は SPE がどのような演算処理をハードウェアで提供しているのかを押さえる程度に軽く流して先へ進み，後々自分でプログラムを書く際にあらためて必要な箇所を読み返すのがよいかと思われます．

第 6 章では，プログラムの SIMD 化や分岐やループの最適化，パイプラインのチューニング

[*11] その中のどこに書かれているのかを探すのはなかなか大変ですが ⋯．

[*12] `http://www.bsc.es/projects/deepcomputing/linuxoncell/cellsimulator/sdk3.1/sources/`

といった，SPE プログラミングにおいて重要となるプログラムの最適化技術を，実例を交えながら紹介します．本書で紹介する SPE プログラミングの最適化コツの大半がここに集約されています．

第 7 章では，これまでの SPE プログラミングを中心とした話題から少し離れ，Cell/B.E. 上の Linux 環境でユーザプログラムから SPE がどのように利用できるようになっているのかについて簡単に紹介します．

第 8 章では，SPE を複数個同時に動かす方法について紹介します．PPE と SPE の間や SPE 同士の間で並列処理を行うには，ここで紹介する仕組みを理解する必要があります．

第 9 章では，SPE の持つチャネルと呼ばれる機能について紹介します．また，チャネルを利用して SPE や PPE の間でメッセージをやり取りするための仕組みについても紹介します．

第 10 章では，SPE の DMA 転送に関して詳しく述べます．DMA 転送は，大量のデータを処理する SPE プログラムを実装する上で欠かすことのできない仕組みです．この章では，SPE の提供する DMA 転送の機能を一通り紹介します．DMA 転送には非常に多くのバリエーションがあり，一部に難解なものも含まれていますが，最初からすべてを理解しようとせずに，まずはどのようなデータ転送が可能であるかということを大まかに押さえておけばよいでしょう．

第 11 章では，元々SPE 1 個で動作していたプログラムを並列化し，さらにデータ転送のチューニングを行う過程を，具体例を用いて紹介します．

第 12 章では，SPE の持つプロブレムステート領域と呼ばれるレジスタについて紹介します．普段はライブラリなどが隠蔽しているレジスタを生で叩くワクワク感と，猛烈な面倒臭さの両方を同時に味わうことができます．

第 13 章では，SPE から PPE 上の関数を呼び出す外部関数呼び出し機構について紹介します．ここでは，まず，標準で用意されている外部関数の実装について解説し，その後，ユーザによる拡張方法についても紹介します．

そして最後の第 14 章では，SPE のイベント処理機構について紹介します．

最後の 3 章は比較的イロモノ的な章となっていますが，この辺りの内容についてまとまった解説をしているドキュメントを見かけたことがないので，実はレアな情報なのかもしれません．

1.4 本書での表記

本書に登場するプログラムのソースコードは，以下のように四角い枠で囲い “リスト” で始まるキャプションをつけて表記します．また，コマンドなどで生成されるファイルの中身を示す場合も同様の表記としています．ソースコードが長いなどの理由で途中を省略するような場合は，“. . .” という表記を間に挟みます．

リスト 1.1 ソースコードの例

```
vec_float4 *s1, *s2, *s3, *d;
...
for (i = 0; i < n; i++) {
    ...
    d[i] = spu_madd(s1[i], s2[i], s3[i]);
    ...
}
```

ライブラリの API や組み込み関数などの定義は，以下のように上側の角が丸くなった枠で囲って表記します．

```
spe_context_ptr_t spe_context_create(unsigned flags,
                                      spe_gang_context_ptr_t gang);
```

コマンドラインからのコマンド入力および実行結果は，以下のように四隅が丸くなった枠で囲って表記します．その際，特にことわりがない限りは，“%” もしくは “#” で始まっている行が入力するコマンドを表し，その他の行はコマンドの実行結果を表すものとします．実行結果が長くて表記しきれない場合は，ソースコードと同様に “...” という表記を間に挟んで，省略されている部分があることを示します．また，コマンドの行の先頭の “%” および “#” はプロンプトです．プロンプトが “%” の場合は，そのコマンドを一般ユーザ権限で実行することを意味し，“#” の場合は，そのコマンドを root 権限で実行することを意味するものとします．

```
% ls -l /spu/spethread-1791-268509248
合計 0K
-r-------- 1 nosuke nosuke      0 2009-07-16 23:26 capabilities
-rw------- 1 nosuke nosuke      0 2009-07-16 23:26 cntl
...
-r-------- 1 nosuke nosuke      4 2009-07-16 23:26 wbox_info
-r-------- 1 nosuke nosuke      4 2009-07-16 23:26 wbox_stat
```

第 2 章
Cell/B.E. と SPE のアーキテクチャ

> われら 7 人は超人オリンピックチャンピオン キン肉マンに挑戦状をたたきつける
> われわれをたおさずにチャンピオンを名のることはゆるさん!!
> — ゆでたまご『キン肉マン』より —

SPE 上で SPE に十分に最適化された処理を実現しようと思った場合，単なるプログラミング言語に対する知識や経験だけでなく，SPE や Cell/B.E. 全体のアーキテクチャ面についての理解も要求されます．本章では，SPE 向けのプログラミングを行う上で押えておきたい Cell/B.E. および SPE のアーキテクチャについて簡単に紹介します．

2.1　アーキテクチャとインプリメンテーション

まず，Cell/B.E. のアーキテクチャについて説明する前に，Cell/B.E. のアーキテクチャと CBEA との関係について簡単に触れておきます．

CBEA は，“Cell Broadband Engine とはこういう構造で，こういう機能を必ず備えており，場合によってはこういう構造やこういう機能を持っていてもよい” といったことを，事細かく定めたアーキテクチャ仕様です．一方，PS3 などに搭載されている Cell/B.E. は，CBEA の仕様を実際にインプリメンテーション (実装) したものになります．したがって，Cell/B.E. のアーキテクチャは CBEA に準拠したものとなりますが，CBEA において “実装は任意” とされている機能の一部が Cell/B.E. では実際に実装されていなかったりするなど，両者が完全に一致するわけではありません．

本書は，今ある Cell/B.E. の SPE を対象としていることから，本章では，絵に描いた餅である CBEA の話ではなく，そのサブセットである実際の Cell/B.E. のアーキテクチャについて紹介します．CBEA の詳細は文献 [1] に記述されていますので，興味がある方はこちらを参照してください．

2.2　Cell/B.E. の全体構成

図 2.1 に Cell/B.E. 全体のブロック図を示します．Cell/B.E. は，PowerPC と命令互換の汎用的な 64bit プロセッサコアである PPE と，独自の命令セットを持ちデータ処理などに特化し

たプロセッサコアである SPE を 8 個[*1]内蔵した，非対称型マルチコアプロセッサです．各プロセッサコア間は，Element Interconnect Bus (EIB) と呼ばれる 128bit 幅 ×4 本の高スループットなリング状のバスで接続されています．各プロセッサコアは EIB に対して送信用の口と受信用の口を持っており，データの送受信を同時に行うことが可能です．また，EIB には，これらプロセッサコアの他に，XDR DRAM 用のメモリインタフェースである XIO と，IO インタフェースである FlexIO の 2 つのモジュールが接続されています．

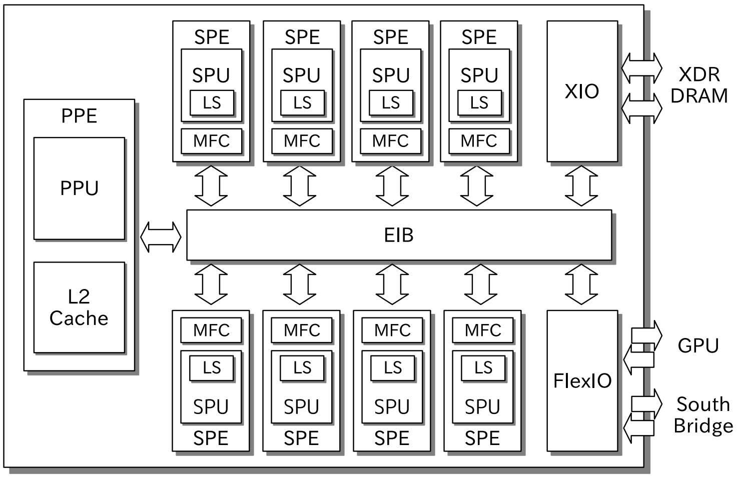

図 2.1　Cell/B.E. のブロック図 (文献 [8] より)

2.3　PPE

PPE は，OS の実行や SPE の管理などを行うことを想定した汎用的なプロセッサコアです．PPE は，内部に PowerPC 970 (いわゆる G5) と命令互換のコアにマルチメディア向け Single Instruction Multiple Data (SIMD) 命令を追加したプロセッサ本体である PowerPC Processor Unit (PPU) や，512KByte の L2 キャッシュなどを備えています．PPU は，PowerPC G5 などと同様にデータ配置としてビッグエンディアンのみをサポートしています．また，PPU はインオーダでの 2 命令同時実行に対応し，2 スレッドの同時実行が可能なハードウェア構成となっています．Intel 系のプロセッサなどと同様に，ソフトウェアからは 2 つの独立したプロセッサコアが存在しているかのように見えます．

PPE の L2 キャッシュはライトバック方式に対応しており，ラインサイズは 128Byte で，8-way セットアソシアティブの構成をとっています．

その他，PPE はハイパーバイザモードなどの仮想化機構を支援する機能をハードウェアで備えており，効率的かつセキュアに複数の OS を同時実行可能であるといった特徴があります．

[*1] 詳細は後述しますが，PS3 に搭載された Cell/B.E. では，8 個のコアのうち 1 個がハードウェア的に使用できないようになっています．また，残りの 7 個のコアのうちの 1 個がハイパーバイザに割り当てられているため，PS3 Linux 上でユーザが Cell/B.E. を利用する場合，ユーザは SPE を 6 個までしか使うことができません．

2.4 SPE

SPE は，マルチメディア向けデータ処理などの演算に特化したプロセッサコアです．SPE は Synergistic Processor Unit (SPU) と呼ばれるプロセッサ本体と，外部とのやりとりを受け持つ Memory Flow Controller (MFC) と呼ばれるモジュールで構成されます．

SPU は内部に Local Storage [*2](LS) と呼ばれる 256KByte の専用メモリを持ち，このメモリ上の命令やデータをアクセスして動作します．SPU は，内部に持っている LS 以外のメモリに直接アクセスすることはできません．256KByte に収まりきらないデータやプログラムを扱う場合は，PPE の主記憶である XDR などがマップされている Effective Address (EA) 空間 (Cell/B.E. のユーザプログラムが利用するアドレス空間 (詳細は後述)) 上の領域と LS 上の領域との間で DMA 転送を行って，データやプログラムの入れ替えを行うことで対応します．DMA 転送自体は MFC が行いますが，MFC に対する DMA 転送の要求は，SPU で動作するプログラム中で明示的に発行する必要があります[*3]．この外部とのデータ転送部分をいかに効率よく設計・実装するかが，SPE プログラムの難しいところの 1 つであり，(好きな人にとっては) 醍醐味でもあります．

なお，SPE の LS はメモリマップド I/O として EA 空間にマップすることができる構造となっており，マップした場合，PPE からは通常のメモリと同じように SPE の中の LS に直接アクセスすることが可能となります．また，EA 空間にマップされた LS は，他の SPE の MFC からも DMA 転送でアクセスできるようになっており，これを利用することで LS 間で直接データ転送を行うことも可能です．

2.4.1 MFC

MFC は，各 SPE に 1 個ずつ内蔵されており，SPU と外部との間の様々なやりとりを受け持ちます．SPU 上で動作するプログラムは，I/O 命令に相当するチャネルアクセス命令を用いることで，同じ SPE 内の MFC に対して指示を出したり，外部からの情報を受け取ったりすることができます．また，PPE や他の SPE からも，EA 空間にマップした MFC の I/O 領域にアクセスすることで，任意の MFC に対して指示を出したりすることができます．

MFC に対する指示の多くはコマンドの形で発行され，MFC に発行されたコマンドは，MFC 上のキューで管理されます．SPU がチャネルアクセス命令を用いて MFC に DMA 転送を要求するコマンドを発行すると，MFC は DMA 転送処理を SPU の処理と独立して行います．

また，MFC は，内部に Synergistic Memory Management (SMM) と呼ばれる MMU を持っており，PPE 上の OS と連携してアドレス変換やメモリ保護を行います．これにより，SPE 上で動作するユーザプログラムは，主記憶などの領域の物理アドレス (実アドレス) を考慮することなく，EA で転送領域を指定して，それらを対象とした DMA 転送を要求することが可能となっています．

[*2] 文献 [5] によると Local Store とも呼ぶそうです．

[*3] 一般的なプロセッサの命令キャッシュやデータキャッシュのように，ハードウェアが勝手にデータを読み書きしてくれたりするような仕組みではありません．

この他，MFC は，SPU と外部との間でメッセージ通信を行うための仕組みや，メモリアクセスの同期を取るための仕組みなどを備えています.

2.4.2 SPU

SPU は，PowerPC などと互換性のない，まったく独自の命令セットを持った SIMD 型のプロセッサです. SPU はマルチメディア向けのデータ処理などの演算を専門に行うことを考慮して設計されているため，一般的なプロセッサと違い，OS などを動かすための機構を備えていません．また，直接アクセス可能なメモリ領域が 256KByte しかない点など，プログラミングを行う上で一般的なプロセッサにはない様々な制約を伴います．PPU と同じく，データ配置としてはビッグエンディアンのみをサポートしています．以下で，プログラミングをする上で押さえておきたい SPU の特徴について簡単に紹介します.

インオーダ実行の 2-way スーパースカラ型プロセッサ

SPU は SIMD 型のデータ処理に対応した (というより演算に関しては SIMD 方式の命令しか持っていない) スーパースカラ型のプロセッサです．SPU は内部に機能の異なる 2 本のパイプラインを持っています．偶数パイプと呼ばれるパイプラインでは，整数演算や浮動小数点演算，論理演算などが行われ，奇数パイプと呼ばれるパイプラインでは，データのロード・ストアや 128bit 幅のシフト・ローテート処理，分岐命令の処理などが行われます.

2 本のパイプラインは並列に動作し，2 つの命令を同時に発行して並列処理させることが可能ですが，SPU はアウトオブオーダ実行には対応していないので，2 命令が同時に発行されるのは，偶数パイプで処理される命令と奇数パイプで処理される命令 (ともに 32bit 長) が 8Byte 単位でアラインしている場合*4に限られます．普通にプログラムを書いているうちはあまり関係がありませんが，プログラムの処理速度を上げたい場合は，いかにして 2 本のパイプラインに命令が詰まった状態を維持させるかを意識する必要が生じます.

SPU はスーパースカラ型？

SPU のように用途がまったく異なった 2 本のパイプラインに対してインオーダでしか命令を発行できないプロセッサをスーパースカラ型と呼んでよいのか疑問に思われるかもしれません．筆者もそう思って不安に感じたのですが，調べてみると世間一般にスーパースカラとして認識されているようなので，本書でもスーパースカラ型として記述しています.

キャッシュを持たない記憶階層

これまで述べたように，SPU には 256KByte の LS と呼ばれるメモリが内蔵されており，LS には SPU が実行する命令列や処理するデータが置かれます．この LS と SPU の間には，キャッシュが存在せず，SPU は直接 LS を読み書きします．これだとデータのロード・ストアと命令の読み出しが競合して性能が落ちそうですが (実際にはこれらに加えて MFC からの DMA 転送に

*4 偶数パイプで処理される命令が 8 で割り切れるアドレスに配置されていて，その直後に奇数パイプで処理される命令が配置されている状態を意味します.

よる読み書きなども競合する)，実は LS は 1 サイクルで 1,024bit ものデータをまとめて読み書きすることのできる脅威の構造になっているようで [9]，これを利用して，命令に関してはまとまった単位でプリフェッチを行うことで，SPU のロード・ストアとのアクセスの競合を回避するようになっています[*5].

128bit 幅 ×128 本の汎用レジスタ

各 SPU は，128bit 幅の汎用レジスタを 128 本持っています．これらのレジスタは整数データだけでなく，単精度や倍精度の浮動小数点数のデータおよびそれらのベクタ (複数のデータを 128bit に詰め込んだもの) などの，SPU の演算で扱えるすべての種類のデータを保持することができます．SPU では比較命令の結果なども専用のフラグではなく汎用レジスタに格納されます．

SPU はハードウェアによる動的なレジスタリネーミングやレジスタウィンドウのようなレジスタの使い方には対応していません．そのため，これら大量のレジスタを有効に活用してプログラムの処理速度を向上させるには，レジスタを大量に使用するような実行コードが生成されるよう，意識してプログラムを記述する必要があります．

なお，SPU には，汎用レジスタ以外のユーザレジスタとして 128bit 幅の FPSCR と呼ばれるレジスタが用意されています．FPSCR は浮動小数点数演算で発生した例外の情報の保持などに用います．

深いパイプラインと静的分岐予測

SPU のパイプラインは非常に段数が深いため，分岐ミスが発生すると 18〜19 サイクルものペナルティが発生します [5]．SPU はハードウェアによる動的な分岐予測器を備えていないことから，何も考えずにプログラムを記述してしまうと，ループの中のちょっとした `if` 文などの条件分岐のせいでパイプラインがスカスカになってしまい，期待したほどの性能が得られないといった状況が発生します．このような状況に対し，SPU では分岐ヒント命令 (分岐の予告のような命令) が用意されており，これを実行コード上で分岐よりも十分手前に入れておくことで，分岐によって発生する性能の低下 (分岐ペナルティ) を抑えられるようになっています．

SPU は，連続したアドレスが逐次実行されるという前提のもと，ある程度のかたまり (32 命令単位) で命令を LS から命令蓄積用のバッファ (命令バッファ) にまとめてプリフェッチしておき，命令バッファから順番に命令を 1 個ないし 2 個取り出して実行します[*6]．命令を実行している途中に分岐ヒント命令が現れると，分岐用のプリフェッチ処理が発動し，分岐ヒント命令に書かれたアドレスから通常のバッファとは別の分岐先専用の命令バッファへの命令のプリフェッチが指示されます．それからしばらくして，実際の分岐命令が命令フェッチの段階で検出されたら，命令のフェッチ元を通常の命令バッファから分岐先の命令バッファに切り替えます．これにより，分岐が期待通り成立した場合に，分岐ペナルティは 0 になります．逆に，不幸にも分岐が不成立だった場合は，分岐命令より前のパイプラインの中身がフラッシュされることになり，分岐ペナルティが発生してしまいます．

[*5] LS へのアクセス優先度は DMA 転送とメモリマップした際の領域への外部からの読み書きが最も高く，その後に SPU からのロード・ストア，分岐ヒント命令による命令のプリフェッチ，通常の命令のプリフェッチと続きます．

[*6] 一見すると命令キャッシュのように思えるかもしれませんが，キャッシュのような命令の再利用性は一切ありません．命令は発行するたびに順次バッファから追い出されていきます．

分岐ペナルティの発生は SPU に限った話ではありませんが，SPU では特に不用意な分岐によるダメージが大きいことから，プログラムの処理効率を上げるためには分岐に気を遣う必要があります．

このように，SPU は，命令のアウトオブオーダ実行やハードウェアによる動的分岐予測，動的なレジスタリネーミングなどの，近年のハイエンドなプロセッサが持つ複雑で強力なハードウェア機構を持たず，これらをすべてコンパイル時の静的な最適化に期待した，割り切った構成のアーキテクチャとなっています．これは，ハードウェアの構造をなるべく単純化することで，回路規模や消費電力を抑えつつ高クロックで動作することを目指したためだと思われます．また，一般的なプロセッサで行われているような，実行コードをコンパクトにまとめたり処理データの配置を工夫したりするなどしてキャッシュのヒット率を上げることで性能向上を図るといったチューニング技法は，SPU には通用しません．将来的にはコンパイラやプログラミングモデルなどの発展により無駄な知識となるかもしれませんが，今現在，SPE の性能を存分に引き出すことのできるプログラムを作成しようと思うなら，これら SPE のアーキテクチャ的な特徴を意識しておく必要がありそうです．

SPE と SPU の呼び分け

本書では，“SPE” と “SPU” という 2 つの名称のどちらも使用します．これらの使い分けはほぼ図 2.1 の通りで，“SPE” といったら MFC も含んだ部分を指し，“SPU” といったら LS とプロセッサ本体を合わせた部分を指します．概ね外 (PPE など) から見てどうこうという話をする場合は “SPE” と呼び，SPE の中の MFC や LS とのやりとりを話題にする場合は “SPU” という呼び方をする傾向がありそうです．ただし，筆者はいい加減なので，どちらでもよい場合は適当にごちゃ混ぜにして使ったりもします．適宜その都度フィーリングで読み取ってください．

2.5 アドレス空間とメモリモデル

これまで述べたように，SPU は，内蔵している LS にしかロード・ストアを行うことができません．MFC との間の情報のやり取りや，SPU 自身の保持している特別な情報へのアクセスは，チャネルと呼ばれる I/O を介して行います．チャネルをアクセスするにはチャネルアクセス命令と呼ばれる専用の命令を用います．チャネルには番号がついており，チャネルアクセス命令では，この番号を指定して読み書きを行います (読んだ結果をレジスタに格納したり，レジスタの値を書き込んだりできる)．また，SPU は，PPE の主記憶や他の SPE の LS，プロブレムステート領域 (ユーザ権限でアクセス可能な SPE のレジスタ) などを含む EA 空間に対しては，MFC に DMA 転送を要求することでアクセスを行います．したがって，各 SPU からすると，LS，チャネル，EA 空間の 3 種類の独立したアドレス空間が存在していることになります．

SPE から見た場合，EA 空間に関しては SMM によってメモリ保護が行われているため，不正なアドレスへのアクセスはできないようになっていますが，一方で，自身の内蔵する LS に対しては，間に MMU などが存在せず，メモリ保護なども一切行われません．また，LS では，デー

タ用のページと命令用のページといった明確な区別もないため，実行中のプログラムをその場で書き換えたり，あるいは元々データとして書き出したアドレスにプログラムカウンタを設定して実行させたりすることが可能です．なお，SPU 自体は 32bit アドレッシングを行うプロセッサなのですが，LS は 256KByte しかないため，実質的にはアドレスの下位 18bit しか有効でなく，上位 14bit は無視されます．

2.5.1 EA

OS やアーキテクチャについて学んだことのある方でも “EA” という言葉は聞き慣れないかもしれません[*7]．PowerPC において，EA は，仮想アドレス空間をさらにセグメンテーション化したアドレス空間を指します．Cell/B.E. において，メモリ保護が機能している状態で，PPE 上で動作しているユーザプログラムが参照するアドレスは EA となります．EA に対してメモリアクセスがあると，EA の下位 28bit を残して上位の変換が行われて仮想アドレスが求められ，その仮想アドレスがさらにページ単位でアドレス変換されて実アドレスになります．Cell/B.E. の EA は 64bit 幅で，それが 65bit 幅の仮想アドレスを経て，最終的に 42bit 幅の実アドレスに変換されます．

なお，通常のユーザアプリケーションを記述する上では，(SPE を利用する場合であっても) 仮想アドレスや実アドレスなどのことを特別気にする必要はありません．ユーザが PPE や SPE のプログラム中で扱うことのできる，主記憶などがマップされているアドレス空間を EA 空間と呼び，通常はそれが 64bit 幅あるということだけ理解しておけば十分です．

2.5.2 メモリコンシステンシモデル

Cell/B.E. のドキュメントを読むと，“Cell/B.E. ではウィークコンシステンシモデルを採用しており，あるプロセッサがメモリにアクセスする順序と，別のプロセッサから観測される順序，そして実際に主記憶で発生するアクセスの順序が異なる可能性がある” といったことが書かれています．これは何を意味するのでしょうか．

たとえば SPE が DMA 転送で PPE の主記憶に LS 上のデータを転送し，その DMA 転送 (データの送出) が少なくとも SPE 上で完了したことを確認した上で，LS 上のフラグを PPE の主記憶に DMA 転送して PPE にデータの準備ができたことを通知する処理を行うケースを考えます．PPE は，フラグが書き込まれるのを主記憶上の領域をポーリングして待ち，フラグの書き込みを検出したら，先に転送されているはずのデータを読み出して処理を行います．が，このとき，メモリコンシステンシがウィークコンシステンシだと，PPE が読み出したデータは，最初に SPE が転送したデータではなく，それ以前にその領域に存在していたデータであるという事態が起こり得ます．すなわち，SPE が行った書き込みの順番と，PPE から見える主記憶へ書き込まれた順番が異なる可能性があり，それが許容されているのです．このような事態の発生により，期待されないデータの受け渡しが発生してしまうのを防ぐために，Cell/B.E. には各所に

[*7] 日本語のドキュメントだと Effective Address は直訳されて “実効アドレス” と書かれていますが，PowerPC や 68000 などの日本語ドキュメントで慣れ親しんでいない方にはピンとこないかもしれないので，本書では “EA” で統一します．

メモリアクセスの同期を行うための機能が用意されています．

しかしながら，本書ではそういった機能は扱いません．なぜなら，少なくとも Cell/B.E. に限れば，ウィークコンシステンシと言いつつ完全にインオーダのコヒーレントなメモリモデルが採用されており，先に述べたような事態は起こり得ないからです．Cell/B.E. に関連する多くのドキュメントでは，メモリコンシステンシ関連の説明がとても難しく書かれており，ある種の初心者キラーとなっているのですが，PS3 の Cell/B.E. でプログラムが期待通り動けばよいのであれば，この辺りのことは気にしなくてよいでしょう．逆に，Cell/B.E. だけでなく，CBEA に準拠したすべてのプロセッサ上で期待通りに動作するようなソフトウェアを開発するのであれば，メモリコンシステンシに関して十分考慮した上でプログラミングを行う必要があるかもしれません．

2.6 実際の Cell/B.E.

ここまで Cell/B.E. のアーキテクチャ的な話をしてきましたが，ここで注意しておくべき点があります．本章で述べたように，Cell/B.E. には 8 個の SPE が搭載されていますが，PS3 などに搭載されている Cell/B.E. は，歩留まりなど (？) の事情により，8 個の SPE のうちの 1 個が最初から物理的に利用できないように処理された状態で出荷されています．そのため，PS3 上のプログラムは，物理的な SPE を 7 個までしか同時に利用することができません．

また，直接ハードウェアとは関係ありませんが，少なくとも Linux を PS3 上で動かしている間は，ハイパーバイザが 7 個の SPE のうちの 1 個を常用することになるため，Linux 上のユーザプログラムからは SPE をさらに 1 個少ない 6 個までしか利用することができません

並列プログラムを書き慣れた人は，同時に使える SPE の数が 2 のべき乗ではないと聞いただけで不便に感じるかもしれませんが，これはどうしようもないので，ここでは発想を変えてプログラムを書くべきでしょう．たとえば従来の並列プログラムのように 1 つの問題を 2^n 等分して個々の計算資源に割り付けて解かせるという考え方にこだわらずに，問題をより細かい単位で細切れに分割して用意しておき，各 SPE は手が空くたびに未処理の問題片を順番に処理していくといったモデルでプログラムを組むなどの方法が考えられます*8.

*8 もちろんそのように分割できればの話ですが.

第 3 章
開発環境の準備

昔…人と武器はひとつだった… 人は強き武器に恥じぬよう努力した…
強き者がいるからこそ武器も日々進歩した…
今はどっちもクズだ…！

— 三条陸・稲田浩司『ダイの大冒険』より —

本章では，本書で紹介する SPE プログラミングを実践するのに必要となる開発環境を構築する手順を簡単に紹介します．本章では，資源を大切に使うということを念頭に，本当に本書で必要となるものしかインストールしません．心配だから何でもかんでも入れておきたいという方は，本章を参考とせずに，Fixstars の Web ページ*1 などを参考にインストールを行ってください．

また，筆者は PS3 上でできあいの Linux ディストリビューションを動かすことそのものにはあまり興味がないため，現在 PS3 でどのような Linux ディストリビューションがどのような形態で動作して，それぞれにどのような特徴があるのかといったことは一切把握できていません．本章では Cell SDK 3.1 がサポートしているディストリビューションである Fedora 9 をインストールしますが，Gentoo などの他のディストリビューションや 1CD ブートタイプの Linux の方がよりよい選択肢だと思われる場合は，そちらをお使いください．

3.1 色々な開発環境の構成

SPE プログラミングを行う上で，PS3 への Linux のインストールは必須となりますが，PS3 Linux (PS3 上の Linux) をどこから操作をして，PS3 Linux 上で実行するファイルをどこでコンパイルをするのかといった部分には色々な選択肢があり，これらの組み合わせによって様々な開発環境の構成が考えられます．たとえば図 3.1 のように，PS3 にキーボードとマウスを接続して，PS3 Linux に Cell SDK をインストールしておけば，PC などを持っていなくてもプログラムの開発を行うことができます．が，PS3 単体で開発を行おうと思うと，PPE が遅いなどの理由から全体的に動作が緩慢な上に，人によっては普段 Windows などで使っているものとまったく違う GUI やエディタを強要されることになるため，非常にストレスがたまることでしょう．よほどの事情がない限り，この構成はおすすめしません．

一方，図 3.2 に示すように，PS3 と普段使っている PC を同じネットワークに接続し，さらに

*1 `http://cell.fixstars.com/ps3linux/index.php/メインページ`

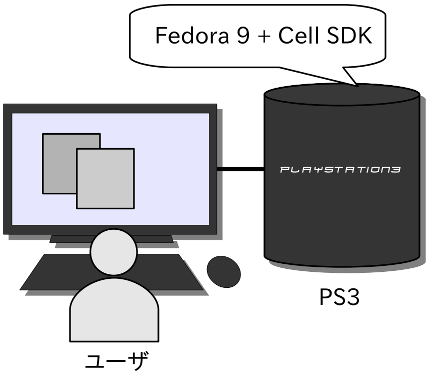

図 3.1　PS3 単体で開発を行う構成

PC 上に VMware Player などの仮想化環境を導入した構成をとる方法も考えられます．この構成で，仮想化環境上にも Fedora 9 と Cell SDK をインストールしておき，コンパイルなどは仮想化環境上の Fedora 9 上で行って，生成された実行ファイルの実行のみを PS3 上で行うようにすることで，ストレスなく開発を行うことが可能となります．ただし，便利な分，セットアップに若干手間がかかるという弱点もあります．

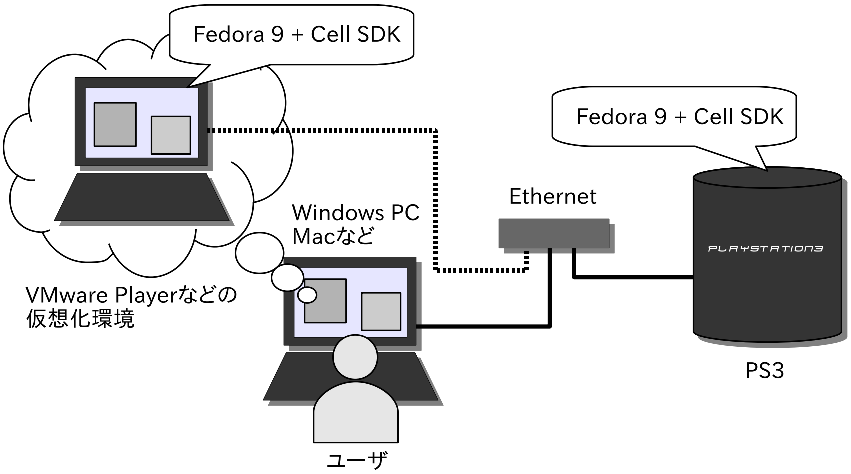

図 3.2　PS3 と PC に仮想化環境を組み合わせて開発を行う構成

そこで，本書では，これら 2 つの間を取るような構成として，図 3.3 に示す構成の環境のセットアップ手順を紹介します．この構成では，PS3 にのみ Fedora 9 と Cell SDK をインストールし，Windows PC や Mac などの使い慣れた端末から SSH などで PS3 Linux にリモートログインしてコンパイルなどを行います．仮想化環境の導入などが不要なため，セットアップの手間が少なくて済み，また，ソースコードの編集は端末側のエディタなどで行えることから，PS3 単体で開発を行う場合に比べてずっと快適に作業をすることができます．

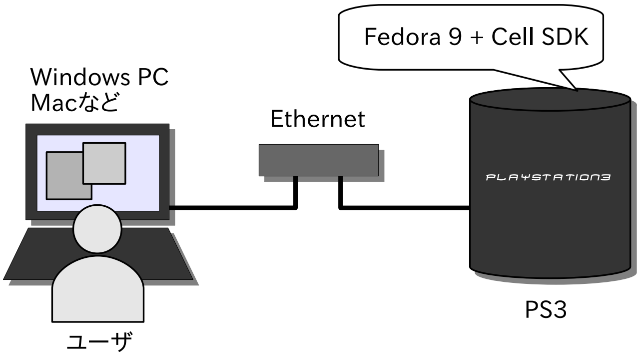

図 3.3　PS3 と PC のみで開発を行う構成

3.2　開発環境の構築に必要となるもの

インストールの際に必要となるものを以下に示します．

- USB キーボード

  一般的なものであれば，どこのメーカーのものでも構いません．セットアップが完了したら不要となるので，一時的にどこかから借りてくるのもよいでしょう．

- Fedora 9 for PPC の DVD

  JAIST の FTP サーバ[*2] などから `Fedora-9-ppc-DVD.iso` をダウンロードして，PC 上で DVD-R か何かに焼きます．そう何度も使うものでもないので，DVD-R ではなく DVD-RW などに焼くのもよいかもしれません．

- PS3 Linux Distributor's Starter Kit の CD

  PS3 Linux の FTP サイト（？）[*3] から `CELL-Linux-CL_20080609-ADDON.iso` をダウンロードして，PC 上で CD-R か何かに焼きます．こちらも何度も使うものでもないので，CD-RW に焼いた方が無駄が少なくて済みます．

これらの他に，PS3 からインターネットに出られるネットワークが必要になります[*4]．また，端末として Windows PC を使用する場合は SSH クライアントを導入しておいてください．PuTTY と WinSCP あたりをインストールしておけば十分です．

[*2] `ftp://ftp.jaist.ac.jp/pub/Linux/Fedora/releases/9/Fedora/ppc/iso/`
[*3] `ftp://ftp.uk.linux.org/pub/linux/Sony-PS3/distro-kit/`
[*4] Fedora 9 をインストールしたままの状態でアップデートせずに利用するのであれば不要ですが ….

3.3　PS3 への Fedora 9 のインストール

最初に，PS3 に Fedora 9 をインストールします[*5]．なお，以下で紹介する手順は，初期型の PS3 (HDD 20GByte モデル) にバージョン 3.1.0 のファームウェアがインストールされた状態での操作を想定しています．ファームウェアのバージョンによってはメニューの選択肢や機能などが異なる可能性があるのでご注意ください[*6]．

3.3.1　ハードディスクの分割とフォーマット

まず，Fedora 9 をインストールするハードディスク領域を用意します．Fedora 9 を外付けハードディスクにインストールすることも可能なようですが，筆者は試したことがないので，ここでは無難に内蔵ハードディスクにインストールすることにして，そのための領域の用意の方法を紹介します．

内蔵ハードディスクにインストールする場合，最初にハードディスクの領域を 2 つに分けて，片方を Linux 用の領域とする必要があります．分割などの処理は，PS3 のゲーム OS が自動で行ってくれます．一応，その際，ユーザ情報くらいは残したまま分割してくれるようなのですが，ダウンロードしたゲームなどのデータはごっそり消えてしまうので，必要に応じてゲーム OS のバックアップ機能などを利用して USB 接続の外付けハードディスクや USB メモリなどにきちんとバックアップを取っておくのがよいでしょう[*7]．

ハードディスクの分割は，クロスメディアバーのメニューを "設定" → "本体設定" とたどって "フォーマットユーティリティ" を選択し，"ハードディスクのフォーマット" を選んで進むことで始めることができます．最初に分割の有無を聞かれるので，ここでは "すべてを PS3 に割り当てる" ではなく "カスタム" を選びます．すると，どのように分割するのかを聞かれます．本書での Fedora 9 のインストール方法であれば 10GByte も要らないのですが，他に選択肢がないので，"他のシステムに 10GB を割り当てる" を選びます．続いてハードディスクのフォーマットをどう行うのか聞かれます．ここは "クイックフォーマット" と "フルフォーマット" のどちらでもよいのですが，フルフォーマットをすると非常に時間がかかるので，特に理由がなければクイックフォーマットでよいでしょう．決定して先に進むとフォーマットが始まります．

3.3.2　ブートローダのインストール

次に，Fedora 9 を起動するために必要となるブートローダをインストールします．ブートローダは PS3 Linux Distributor's Starter Kit に含まれています．PS3 Linux Distributor's Starter Kit (`CELL-Linux-CL_20080609-ADDON.iso`) を焼いた CD を PS3 に挿入し，クロスメディアバーのメニューから "設定" → "本体設定" とたどって "他のシステムのインストール" を選択します．先に進むと "以下のインストールデータがみつかりました" と表示され，その下に

[*5] もっと新しい Fedora のリリースがあるのに何で？と思われるかもしれませんが，単に最新の Cell SDK である Cell SDK 3.1 が Fedora 9 を前提として作られているためです．

[*6] 2009 年 12 月の時点での最新型の PS3 には Linux をインストールすることができません．ご注意ください．

[*7] 大切なセーブデータが消えてしまっても筆者は一切責任を負いかねますので，あらかじめご了承ください．

“/PS3/otheros/otheros.bld” というパスが表示されるはずです．ここで “開始する” を選ぶとインストールが始まります．インストールが完了したら CD を取り出しておきます．

3.3.3 ブートローダの起動

続いてブートローダを起動します．まず，PS3 に USB キーボードを接続し，Fedora 9 の DVD を PS3 のドライブに入れます．次に，クロスメディアバーのメニューを “設定” → “本体設定” とたどり，“優先起動システム” を選択します．これを選ぶと “PS3” と “他のシステム” が選択肢として表示されるので，“他のシステム” を選んでください．すると，他のシステムを起動させるかどうか聞かれるので，“はい” を選択します．選択後，PS3 が再起動すると，ゲーム OS が起動せずに 2 匹のペンギン (Tux) が画面に表示され，“kboot:” とプロンプトが表示されるはずです*8．

3.3.4 Fedora 9 のインストール

そしていよいよ Fedora 9 のインストールを行います．よくあるインストールの説明だと，“kboot:” と表示されている後ろに “linux64 video=720p” と入力するよう書かれていたりしますが，これだとゴージャスな X のインストーラが起動して大変鬱陶しいので，ここではあえて “linux64 text” と入力します．これで X を使わないキャラクタベース (といっても AA のように文字を駆使してダイアログなどが表示されるようにはなっている) のインストーラが起動します．

“linux64 text” でインストーラを起動した場合，操作はすべてキーボードで行うことになります．キーボードのタブを押すたびに，フォーカスがボタンや文字入力エリアの間を移動します．チェックができそうな項目がある場合は，そこにフォーカスを移してスペースを押すとチェックができます．メニューから選べるような構造の部分では，そこにフォーカスを移して矢印キーでメニューを動かすことができます．ボタンを押す場合も，そこにフォーカスを移して Enter かスペースを押します．

インストーラが起動すると，最初にディスクのチェックをするかどうかを聞いてきます．ここで “OK” を選んでしまうと無駄に待たされることになるので，“Skip” を選んで飛ばしましょう．

次に，インストール中に使用する言語やキーボードのキー配置を聞かれます．言語の選択肢の中には日本語もあるのですが，このモードでは日本語は使用できません (選ぶと強制的に英語になる)．ここは大人しく “English” を選んでおきましょう．一方，キーボードのキー配置では，日本語キーボード (“jp106”) を選択することも可能です．お使いの環境に合わせたものを選んでおきましょう．

続いて細々とした設定を行います．最初に，ネットワークの eth0 を設定するかどうかを聞かれます．“Yes” を選び，IPv4 を有効にして，使用しているネットワーク環境に合わせて DHCP を有効にするなどの設定を行います．次にタイムゾーンなどの設定を行います．ここでは UTC

---

*8 筆者もこの辺りのことはさっぱりわかっていないのですが，どうも PS3 のブートローダー自体が Linux の上で動くようになっているようです．この辺りの仕組みに関してはとても興味があるのですが，残念ながらリソース不足によりケアできていません．

を使う設定にしたまま，タイムゾーンとして“Asia/Tokyo”を選びましょう．続いて root のパスワードを設定します．脆弱なのをつけると怒られますが，その辺りは運用形態と照らし合わせて適宜判断してください．また，日本語キーボードを英語レイアウトで使用している場合は，記号を入力する際に注意してください*9．

細かな設定が完了したら，いよいよディスクのフォーマットに移ります．色々と構成に凝るのもよいのですが，面倒なのでここはすべてデフォルトのまま先に進みます．進んでいくとフォーマットが開始するので，しばらく待ちましょう．

フォーマットが済んだらインストールするパッケージを選択します．こちらは面倒臭がって全部入れようだなんて思ってはいけません．貴重なハードディスク領域が本当にどうでもよいもので埋め尽くされてしまうので，ここでは手間を惜しまず**何も入れない**ようにします．“Office and Productivity”のチェックを外した上で，“Customize software selection”にチェックをつけて先に進みます．すると，詳細なパッケージ群を選択するよう求められるので，ここではすべてのチェックを外します．これで先に進むとインストールが始まります．インストールが終わるまで結構時間がかかるので，お茶でも飲みながらのんびり待ちましょう．この間に 3.4 で用いるファイル類をダウンロードしておくのもよいかもしれません．

パッケージのインストールが完了すると，DVD が排出され，PS3 を再起動するよう促されるので再起動します．これで Fedora 9 のインストールは完了です．

3.3.5　Fedora 9 のセットアップ

最後に，インストールした Fedora 9 のセットアップを行います．インストールしたままの状態だと，無関係なサービスが動いていたり，不要な機能が有効になっていたりして，リソース (特にメモリ) をかなり食っているので，ここではこれを適当に止めることを中心に行います．

インストール後，PS3 が再起動すると，再びブートローダが表示されます．今度は 5 秒くらい触らないでいると勝手に Fedora 9 が起動します．Fedora 9 が起動したら，ログインプロンプトが表示されるので，root アカウントでログインします．ログインしたら `ifconfig` コマンドで eth0 の IP アドレスを調べましょう*10．

IP アドレスが確認できたら PS3 の前から離れ，PC 上の SSH クライアントで，先ほど調べた PS3 の IP アドレスを入力して，PS3 に SSH で接続します．PS3 に向かった作業 (苦行？) はここでおしまいです．ここからは使い慣れた PC の上で，SSH クライアントを利用して作業を行います．

不要なサービスの停止

SSH で PS3 に接続して root でログインしたら，最初に不要なサービスを止めます．

まず `setup` コマンドを実行します．画面が文字化けなどにより崩れる場合は，以下のように環境変数 `LANG` を `C` に設定しつつ `setup` を実行してください．

*9 とりあえずここでは適当にアルファベットと数字でつけておいて，後ほどセットアップ中に `passwd` コマンドでつけ直した方が楽かもしれません．

*10 DHCP で IP アドレスが取れる環境であれば，ブートローダの画面で `kboot:` プロンプトの上に IP アドレスが表示されるので，そこで出た値を見て確認してもよいでしょう．

```
# LANG=C setup
```

再びテキストベースのメニューのようなものが出てきます．最初に “Firewall configuration” を選んで，“Disable” にチェックを入れておきましょう．次に “System services” を開いて，要らなそうなものを切っておきます．よくわからない場合は，以下を残して，それ以外のチェックをすべて外せばよいでしょう．NFS を使いたくなった場合などは，適宜その都度必要なサービスを有効にしてください．

- NetworkManager
- anacron
- atd
- crond
- irqbalance
- messagebus
- network
- rsyslog
- sshd
- udev-post

SELinux の無効化

次に SELinux を切ります．vi などのエディタで/etc/sysconfig/selinux を開き，“SELINUX=enforcing” と書かれた行を “SELINUX=disabled” に変更して保存します．vi が嫌い・苦手という場合は，以下のようにコマンドを入力して sed で置換してしまいましょう．

```
# sed -i -e 's/^SELINUX=enforcing/SELINUX=disabled/' /etc/sysconfig/selinux
```

IPv6 の無効化

続いて IPv6 を無効化しておきます．以下のように “/etc/modprobe.d/disable-ipv6” を作成して，中に “install ipv6 /bin/true” と書いておくのが最近のやり方のようです．

```
# echo "install ipv6 /bin/true" > /etc/modprobe.d/disable-ipv6
```

ここまでやったら reboot コマンドで一旦再起動します．

Fedora 9 のアップデート

再起動後，再度 SSH クライアントから root アカウントでログインして OS のアップデートを行います．

```
# yum -y update
```

1 回目はアップデートに必要となるファイルのみが更新されるため，一瞬で終わります．終わったら，もう 1 回上記と同じコマンドを入力してアップデートを行います．すると，今度は大量のパッケージのダウンロードとインストールが行われるはずです．これが終わったらアップデートは完了です．

一般ユーザアカウントの追加

今後の作業のために，ここで一般ユーザのアカウントも作成しておきましょう．たとえば “nosuke” というアカウントを作り，パスワードを設定するならば，以下のように入力します．

```
# adduser nosuke
# passwd nosuke
Changing password for user nosuke.
New UNIX password: (ここでパスワードを入力)
Retype new UNIX password: (ここでパスワードを再入力)
passwd: all authentication tokens updated successfully.
```

シャットダウン

Linux 上の操作で PS3 の電源を切るには，コマンドラインから root 権限で `poweroff` コマンドを入力します．しばらくするとシャットダウン処理が始まり，勝手に電源が切れます．無事電源が切れることが確認できたら Fedora 9 のセットアップは完了です．お疲れ様でした．

なお，ここで紹介したのは Linux や Fedora 9 固有の設定・管理方法のほんのさわりでしかありません．より詳細な情報はとてもここには書き切れないので，必要に応じて適宜ネットで検索するなどして調べてください．

3.3.6 Fedora 9 とゲーム OS の切り替え

Fedora 9 からゲーム OS に戻したい場合は，Linux 起動後にコマンドラインから root 権限で `ps3-boot-game-os` コマンドを実行するか，あるいは Linux 起動前に `kboot:`プロンプトが出ているところで `game` と入力して Enter を押してください．逆にゲーム OS から Fedora 9 に戻したい場合は，3.3.3 で示した手順で優先起動システムを切り替えてください．

3.4 PS3 への Cell SDK のインストール

次に，PS3 上に Cell SDK をインストールします．ここでも余計なものは極力インストールしない主義で進めます．もっと便利なツールを知っているという場合や，せっかくだから全部入れたいという場合は，違う方法をとるとよいでしょう．

なお，以下のダウンロードサイトなどの情報は 2009 年 11 月現在のものです．いつの間にか URL が変わったり，古いファイルが消されたりしている可能性があるので (特に IBM のサイト)，リンク切れしている場合は適宜検索エンジンなどで探すようにしてください．

3.4.1 ファイルのダウンロード

IBM のサイトからのダウンロード

最初に PC などで IBM のサイト[*11] に接続し，Cell SDK 3.1 をダウンロードします．“`Download`” タブを選んだ先にある “`SDK downloads`” というカテゴリの “`Developer package`” 欄にある “`Download SDK 3.1 for Fedora 9 from developerWorks`” を選んで，

[*11] `http://www.ibm.com/developerworks/power/cell/`

アンケートに答えるとダウンロード画面に進むことができます*12．ダウンロード画面では，デフォルトで “Download Director” なるよくわからない仕組みを使うようになっていますが，“Download using http” と書かれたタブをクリックして画面を切り替えることで，普通にダウンロードできるようになります．ここでは “CellSDK-Devel-Fedora_3.1.0.0.0.iso” と “CellSDK-Extras-Fedora_3.1.0.0.0.iso” をダウンロードしてください．以上で IBM のサイトからのダウンロードは完了です．なお，これら ISO イメージはループバックマウントしてインストールするので，早まって CD に焼かないでください (もったいないので …)．

BSC のサイトからのダウンロード

次に，BSC のサーバ*13 から GCC などのオープンソース系のツールやライブラリをダウンロードします．まず ppc64 ディレクトリに移動して，以下のパッケージをダウンロードしてください．

- libspe2-devel-2.2.80-132.ppc.rpm
- libspe2-devel-2.2.80-132.ppc64.rpm
- libspe2-2.2.80-132.ppc.rpm
- libspe2-2.2.80-132.ppc64.rpm
- ppu-gcc-4.1.1-166.ppc.rpm
- ppu-gcc43-4.3.2-2.ppc.rpm
- spu-gcc-4.1.1-166.ppc.rpm
- spu-gcc43-4.3.2-2.ppc.rpm
- spu-newlib-1.16.0-17.ppc.rpm
- ppu-gdb-6.8.50-13.ppc.rpm
- spu-gdb-6.8.50-13.ppc.rpm
- ppu-binutils-2.18.50-21.ppc.rpm
- spu-binutils-2.18.50-21.ppc.rpm

ダウンロードが済んだら，1 つ上のディレクトリに戻って，今度は cbea ディレクトリに移動し，elfspe2-2.2.80-132.ppc.rpm というパッケージをダウンロードしてください．以上で BSC からのダウンロードはおしまいです．

3.4.2 Fedora 9 のパッケージの追加

続いて，Cell SDK の各種パッケージをインストールする際に依存が生じて必要になる Fedora 9 のパッケージをあらかじめインストールしておきます．

PS3 の Fedora 9 上で以下のコマンドを実行してパッケージを入れてください．

```
# yum -y install glibc-devel glibc-devel.ppc64
# yum -y install compat-expat1 compat-expat1.ppc64
# yum -y install gmp
```

*12 ダウンロードにはなぜか IBM ID なる ID が必要になるので，持っていない場合はサクッと作ってしまいましょう．

*13 http://www.bsc.es/projects/deepcomputing/linuxoncell/cellsimulator/sdk3.1/CellSDK-Open-Fedora/

3.4.3 Cell SDK のインストール

これでようやく Cell SDK のインストールを始めることができます．

まず，先ほど BSC のサーバからダウンロードしたパッケージ類をまとめて SCP などで PS3 の Fedora 9 のディスク上にコピーします．そして，PS3 の Fedora 9 上でコピー先のディレクトリに移動して，rpm コマンドを実行してまとめてインストールします．

```
# rpm -ivH *.rpm
```

先ほどの yum コマンドできちんとパッケージが追加できていれば，エラーが出ることなく BSC からダウンロードした全パッケージのインストールが完了するはずです．

次に，IBM のサイトからダウンロードした Cell SDK 3.1 の ISO イメージに含まれるファイルをインストールします．先ほどダウンロードした 2 つの ISO イメージを PS3 の Fedora 9 のディスク上にコピーします．コピーが済んだら，まず CellSDK-Devel-Fedora_3.1.0.0.0.iso の方に含まれるファイルからインストールを行います．以下のようにして ISO イメージをループバックマウントし，その中の rpms 以下に含まれる spu-simdmath-devel-3.1-1.ppc.rpm というパッケージをインストールします[*14]．インストールが完了したらアンマウントします．

```
# mkdir -p /mnt/cdrom
# mount -o loop CellSDK-Devel-Fedora_3.1.0.0.0.iso /mnt/cdrom
# rpm -ivH /mnt/cdrom/rpms/spu-simdmath-devel-3.1-1.ppc.rpm
# umount /mnt/cdrom/
```

続いて CellSDK-Extras-Fedora_3.1.0.0.0.iso のファイルのインストールを行います．同じように ISO イメージをループバックマウントして，rpms 以下の cell-spu-timing-3.1-2.ppc.rpm というパッケージをインストールします．インストールが完了したらアンマウントしておきます．

```
# mount -o loop CellSDK-Extras-Fedora_3.1.0.0.0.iso /mnt/cdrom
# rpm -ivH /mnt/cdrom/rpms/cell-spu-timing-3.1-2.ppc.rpm
# umount /mnt/cdrom/
```

本書で必要となる開発環境の準備は以上で完了となります．お疲れ様でした．

なお，本章でインストールした Cell SDK のパッケージの各種コマンドは，/usr/bin 以下に配置されるため，コマンドを利用するために別途パスを追加する必要はありません[*15]．

[*14] 400MByte 以上もあるイメージをダウンロードしてこれしか使わないのかと思われるかもしれませんが，残念ながら本書ではこれしか使いません．

[*15] spu_timing のみ/opt の下のパスの通っていないところにインストールされますが，このコマンドをコマンドラインから直接呼び出すことはまれなので，わざわざパスを通すこともないでしょう．

第 4 章
SPE を 1 個だけ使うプログラム

フフ … オレはあいさつ仮面シリーズに生き方を教わったようなものだ …!
好きとかどうとかのレベルではない！
— うすた京介『セクシーコマンドー外伝 すごいよ!!マサルさん』より —

本章では，SPE を 1 個だけ使ったプログラムを実際に書いて動かしてみたいと思います．動かすものは何のひねりもない Hello World なのですが，それを SPE 上で動かすには，色々と手続きを踏む必要があります．ここでは，どの関数をどのような順番で呼べばよいのかということを天下り的に覚えるのではなく，それぞれが何をしていて，なぜこの順番で呼ぶ必要があるのかということを理解しましょう．

4.1　SPE のプログラムを呼び出す流れ

第 2 章でも述べましたが，Cell/B.E. には SPE の他に PPE が搭載されており，OS やユーザアプリケーションは通常 PPE の上で動作します．Hello World に限らず，Cell/B.E. で何らかの SPE 用のプログラムを実行する場合は，SPE のプログラムをハードディスクなどの二次記憶から主記憶にロードして，それを LS にコピーし，実際に SPE のプログラムカウンタをセットして実行を開始させる処理を行う PPE のプログラムが別途必要になります．SPE 上でプログラムを実行する上で必要となるこれらの各種処理は，Cell SDK では libspe2 と呼ばれるユーザライブラリが提供しています．PPE 上のユーザプログラムが libspe2 を用いて SPE のプログラムを実行する際の流れを図 4.1 に示します．

ループや条件分岐のない非常にシンプルなフローチャートですが，実際に SPE のプログラムを動かして終わるのを待つだけのプログラムは，この図のような流れとなります．図 4.1 の流れを簡単に説明すると，PPE 上のユーザプログラムは，まず HDD などの二次記憶から SPE の実行ファイルをロードし，次に SPE で動かすコンテキストを生成します．そして生成したコンテキストに実行ファイルの実行コード部分を割り付け，その後，SPE にコンテキストの実行を開始させます．SPE のプログラムが終了したら，コンテキストを破棄して実行ファイルを閉じます[*1]．コンテキストという言葉に馴染みのない方もいるかもしれませんが，ここでいうコンテキストとは，SPE の各種状態 (実行するプログラムやメモリの内容，レジスタの値など) をひとま

[*1] 実行ファイルは SPE のコンテキストに実行コードの割り付けを行った時点で不要となるため，後々別のコンテキストで再利用しないのであれば，もっと早い段階で閉じることも可能です．

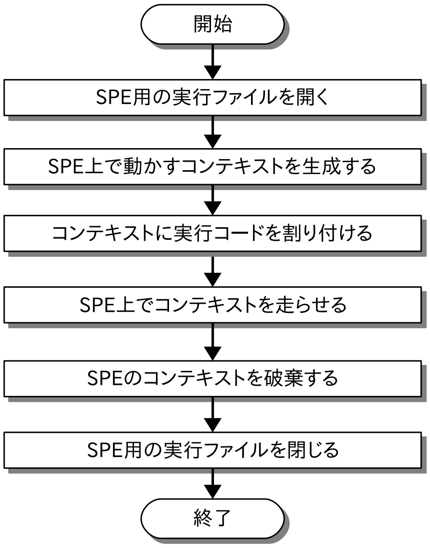

図 4.1　libspe2 を使った SPE プログラム実行の流れ

とめにして管理するものと認識しておけばよいでしょう．

また，余談となりますが，Linux では SPE の処理をコンテキストの形で扱い，実際の SPE 上のコンテキストを PPE が適宜切り替えることで，物理的に存在する SPE の数以上のコンテキストを時分割で同時に動かすことに対応しています．たとえば PS3 Linux ではユーザプログラムからは物理的に 6 個の SPE しか利用することができませんが，一度に 7 個以上のコンテキストを生成して実行することが可能です．もちろん，その場合に同時に活性化できるコンテキストは 6 個までに限られます．複数のコンテキストを生成して同時に実行する方法に関しては第 8 章で詳しく紹介します．

4.2　SPE のプログラムを呼び出す API

実際にプログラムの例を示す前に，図 4.1 に示した個々の処理を行うための libspe2 の API を紹介しておきます．いきなり API だけ見せられてもわけがわからないかもしれませんが，その場合はとりあえず流して読んで，後々ソースを紹介する際に付け合わせながら見てください．なお，以下の API を利用するには，`libspe2.h` を `include` しておく必要があります．また，libspe2 の API の詳細については文献 [4] に記載されています．以下で紹介するものも含めてより深く知りたい場合は，あわせて参照するとよいでしょう．

spe_image_open：SPE の実行ファイルを開く

```
spe_program_handle_t *spe_image_open(const char *filename);
```

`spe_image_open` は，`filename` のパスにある SPE の実行ファイル (ELF ファイル) を開いて中身を PPE の主記憶上に読み出します．戻り値としてプログラム本体や付随する情報を含んだ `spe_program_handle_t` 型の構造体のアドレスを返します．この戻り値は，この関数より後で

実際に SPE の上でプログラムを実行する際に用います．本関数でエラーが発生した場合，NULL が返り，errno にエラーを示す番号が入ります．

spe_context_create：SPE のコンテキストを作る

```
spe_context_ptr_t spe_context_create(unsigned flags,
                                     spe_gang_context_ptr_t gang);
```

spe_context_create は，SPE のコンテキストを新規に生成して返します．戻り値の型である spe_context_ptr_t は，SPE のコンテキストを保持する struct spe_context 型の構造体のポインタを typedef した型になります．本関数を呼ぶと，ユーザライブラリ内でコンテキストに必要な変数などが確保されます．

第 1 引数の flags には，コンテキストに設定する様々な属性をフラグとして渡します．たとえば SPE_MAP_PS というフラグを設定すると，ユーザ権限でアクセスすることが許されるレジスタ (プロブレムステート領域) が EA 空間上にマップされるようになります．

第 2 引数の gang は，ギャングスケジューリングを行う際に必要となる引数です．本書ではギャングスケジューリングについては扱いませんので，ここは常時 NULL としておきます．

本関数でエラーが発生した場合，NULL が返り，errno にエラーを示す番号が入ります．

spe_program_load：SPE のプログラムを LS にロードする

```
int spe_program_load(spe_context_ptr_t spe, spe_program_handle_t *program);
```

spe_program_load は，spe で示される SPE の LS に program が指す実行ファイルの実行コード部分をロードします．program は spe_program_handle_t 型の構造体のポインタで，通常は先ほど紹介した spe_image_open の戻り値を渡します．本関数は成功したら 0 が返り，エラーの際は −1 が返って errno にエラーを示す番号が入ります．

spe_context_run：SPE のコンテキストを実行する

```
int spe_context_run(spe_context_ptr_t spe, unsigned *entry, unsigned runflags,
                    void *argp, void *envp, spe_stop_info_t *stopinfo);
```

spe_context_run は，spe で示される SPE のコンテキストを SPE 上で実行することを指示します．entry は SPE のプログラムのエントリポイント (SPE が実行時に最初に実行する命令) の LS 上のアドレスを指定します．SPE_DEFAULT_ENTRY というマクロ名を指定しておくと，SPE の実行ファイルの ELF ヘッダから得られるエントリポイントの情報がそのまま使われるようになるので，通常はこれを指定します．

runflags には，コンテキスト実行時のオプションをフラグで指定します．フラグの値によって SPE への引数の渡し方などを変更できます．通常は特にフラグを指定する必要はありませんので 0 を渡します．

argp と envp は SPE の main の引数になります．両者とも void *型のポインタなので，呼び出し元の PPE の実行ファイルが 32bit バイナリか 64bit バイナリかによってデータの幅が変

わってきますが，SPE にはどちらの場合も 64bit 幅の符号なし整数として渡されます．引数が必要なければ，これらに NULL を与えることもできます．

stopinfo は SPE のプログラムが終了した際に終了の原因などの情報が入る spe_stop_info_t 型の構造体のアドレスを指定します．spe_context_run から戻った際に SPE のプログラムが終了した理由を確認する必要がなければ，ここは NULL を設定しておけばよいでしょう．

なお，本関数はブロッキング型の関数であり，**SPE 上のプログラムが何らかの理由で終了するまで戻りません．**本関数は，正常に終了すると 0 以上の数が返り，エラーが発生した場合は −1 が返って errno にエラーを示す番号が入ります．戻り値が 0 の場合，SPE 上のプログラムが exit できちんと終了したことを意味します．1 以上の値の場合は，SPE 上のプログラムが stop-and-signal と呼ばれる停止命令で動作を停止したことを意味し，戻り値として stop-and-signal の引数が得られます．ただし，1 以上の値が戻るのは，本関数の runflags に SPE_NO_CALLBACKS というフラグをセットしている場合に限られます．通常は，SPE 上のプログラムが stop-and-signal を呼んで停止した場合でも，spu_context_run 内部で自動的に SPE の処理を再開させるため，本関数から戻ることはありません．stop-and-signal に関しての詳細は第 13 章にて紹介します．

spe_context_destroy：SPE のコンテキストを破棄する

```
int spe_context_destroy(spe_context_ptr_t spe);
```

spe_context_destroy は，spe で示される SPE のコンテキストを破棄します．本関数は成功したら 0 が返り，エラーの場合は −1 が返って errno にエラーを示す番号が入ります．なお，SPE のコンテキストは PPE のプログラム終了時に勝手に破棄されるので，本関数を呼ばずに PPE のプログラムを終了させても特に問題はありません．

spe_image_close：SPE の実行ファイルを閉じる

```
int spe_image_close(spe_program_handle_t *program);
```

spe_image_close は，program の指す spe_image_open で開いた SPE の実行ファイルを閉じます．本関数は成功したら 0 が返り，エラーが発生した場合は −1 が返って errno にエラーを示す番号が入ります．

spe_context_destroy と同様に，本関数を呼ばずに PPE のプログラムを終了させても問題ありません．

4.3 Hello World

先に長々と API の解説をしましたが，ここからは実際に SPE 用のプログラム (Hello World) と，それを実行するのに必要となる PPE 用のプログラムを例に，SPE のプログラムの動かし方を紹介していきます．

とりあえず SPE 上で動くプログラムから用意しましょう．リスト 4.1 のソースを打ち込んで，`hello_spe.c` と名前をつけて保存してください．

リスト 4.1 hello_spe.c

```
#include <stdio.h>

int
main(unsigned long long spe, unsigned long long argp, unsigned long long envp)
{
    printf("Hello World\n");
    return 0;
}
```

保存したら，以下のようにコマンドを入力してコンパイルします．コンパイルには SPU (SPE) 用の GCC である `spu-gcc` を用います．最適化オプションは何でもよいのですが，ここでは`-O3` としています[*2]．正しくコンパイルが通ると `hello_spe` という実行ファイルができるはずです．

```
% spu-gcc -O3 -Wall hello_spe.c -o hello_spe
```

`hello_spe` ができたら，今度はリスト 4.2 のコードを別のファイルに打ち込んで `hello_ppe.c` と名前をつけて保存してください．

リスト 4.2 hello_ppe.c

```
#include <stdio.h>
#include <libspe2.h> // libspe2でSPEを制御する場合は必ずinclude

int
main(int argc, char *argv[])
{
    spe_program_handle_t *program;
    spe_context_ptr_t context;
    unsigned entry = SPE_DEFAULT_ENTRY;

    // SPEプログラム（同じディレクトリにあるhello_spe）を開く
    if ((program = spe_image_open("hello_spe")) == NULL) {
        perror("spe_image_open");
        return -1;
    }
    // SPEコンテキストを作る
    context = spe_context_create(SPE_MAP_PS, NULL);
    // SPEコンテキストにhello_speの中身をロード
    spe_program_load(context, program);

    printf("start SPE program\n");
    // SPEコンテキストを実行
    spe_context_run(context, &entry, 0, NULL, NULL, NULL);
    printf("done\n");

    // SPEコンテキストを破棄
    spe_context_destroy(context);
    // SPEプログラムを閉じる
    spe_image_close(program);

    return 0;
}
```

*2 本書ではこの先も特に記述がない限り`-O3` を `spu-gcc` の最適化オプションとして常時つけるものとします．

保存したら以下のようにコマンドを入力してコンパイルします．コンパイルには PPU (PPE) 用の GCC である ppu-gcc を用います．PPE のプログラムでは，libspe2 の API を呼んでいるので，コンパイルの際に-lspe2 をオプションとして与えて libspe2 とリンクさせます[*3]．

```
% ppu-gcc -O2 -Wall hello_ppe.c -o hello_ppe -lspe2
```

コンパイルが完了したら，hello_ppe を PS3 Linux 上で実行してみましょう．正しく入力できていれば，以下のような実行結果となるはずです．

```
% ./hello_ppe
start SPE program
Hello World
done
```

4.3.1 コンパイラと実行ファイルについての解説

それではここまでの内容について説明します．先にコンパイル作業について説明します．既に述べた通り，ppu-gcc と spu-gcc は，それぞれ PPU (PPE) 用の実行ファイルと SPU (SPE) 用の実行ファイルを生成するコンパイラ (を呼び出すコマンド) です．hello_ppe を file コマンドで確認すると，以下のように表示されるはずです．

```
% file hello_ppe
hello_ppe: ELF 64-bit MSB executable, 64-bit PowerPC or cisco 7500,
version 1 (SYSV), dynamically linked (uses shared libs),
for GNU/Linux 2.6.9, not stripped
```

ppu-gcc はデフォルトで 64bit バイナリを生成していることがわかります[*4]．

一方，hello_spe は file コマンドで確認すると以下のように表示されるでしょう．

```
% file hello_spe
hello_spe: ELF 32-bit MSB executable, Cell SPU, version 1 (SYSV),
statically linked, not stripped
```

SPE プログラムの実行ファイルは 32bit バイナリとして扱われます．したがって (というと語弊があるかもしれませんが)，SPE の C 言語のソース中で long 型の変数を宣言した場合，32bit 幅のデータ型として扱われることになります．また，ポインタも 32bit 幅となります．

4.3.2 PPE のプログラムの解説

PPE のプログラムは，4.2 で紹介した API を 4.1 で述べた順番で呼び出しているだけのプログラムです．各 API の意味や引数などについては，既に 4.2 にて詳しく紹介しているので，ここでは説明を省きます．

[*3] ppu-gcc の方の最適化オプションは-O2 としていますが，これは-O3 より-O2 の方が速いプログラムが生成される気がするという筆者の経験に基づいています．PPE に関しては真面目に調べたことがほとんどないので，実際どちらがよいのかはよくわかりません．

[*4] 32bit バイナリを生成したい場合は，コンパイルの際に-m32 オプションをつけるか，あるいは ppu-gcc の代わりに ppu32-gcc を利用します．

なお，本書では，誌面の都合や読み易さのことを考えて，libspe2 の API の戻り値のエラーチェックをほとんど行っていません．ここで示した例でも `spe_image_open` 以外の関数については一切戻り値の確認を行っていません．実際に自分でプログラムを書く際は，これを真似せずに，きちんとエラーチェックを行うことをおすすめします．

gcc と gcc43

Cell SDK 3.1 でインストールされる SPE 向けの GCC のコマンドには，`spu-gcc` と `spu-gcc43` の 2 種類があります．また，PPE 向けの GCC のコマンドについても，`ppu-gcc` と `ppu-gcc43` の 2 種類がインストールされます．これらはともにバージョンの異なるコンパイラを呼び分けるために用意されています．末尾に “43” がついている方のコマンドは GCC 4.3.2 をベースとしたコンパイラを呼び出すコマンドで，そうでない方は GCC 4.1.1 をベースとしたコンパイラを呼び出すコマンドとなっています．本書で紹介する内容は，どちらのコンパイラを利用してもコンパイル可能ですが，GCC 4.3.2 ベースのコンパイラの方が効率のよい実行コードを生成する傾向があるようなので，プログラムの実行速度を上げたい場合は後者を利用するのがよいでしょう．

4.3.3 SPE のプログラムの解説

次に SPE のプログラムについて説明します．まずはリスト 4.3 に示す `main` の関数宣言部分に着目してみましょう．

リスト 4.3 hello_spe.c の main 関数の引数

```
int
main(unsigned long long spe, unsigned long long argp, unsigned long long envp)
{
```

一見するとごく普通の `main` 関数ですが，よく見ると引数の型と個数が少し違っています．これら 3 個の引数には以下の値が入ります．

- 第 1 引数 `spe`: PPE 上の SPE のコンテキストのアドレス
- 第 2 引数 `argp`: PPE で `spe_context_run` に渡した第 4 引数の値
- 第 3 引数 `envp`: PPE で `spe_context_run` に渡した第 5 引数の値

4.1 で述べたように，`spe_context_run` の第 4 引数と第 5 引数が `main` 関数の第 2 引数と第 3 引数に渡ります．文献 [4] によると，`spe_context_run` の第 4 引数と第 5 引数は，それぞれアプリケーション固有のデータと環境固有のデータを指すとあるので，本来は SPE が実行開始直後にこれら引数をアドレスとみなして，DMA 転送で実行時に必要となるデータを読み出すのに用いたりするのでしょう．今回の PPE のプログラムでは `NULL` を渡しているため，`argp` と `envp` には 0 が入ります．

また，第 1 引数には `spe_context_run` の第 1 引数 (コンテキストのポインタ) が渡されるのですが，コンテキストのポインタを渡されたところで，SPE 側で特に使い道があるわけでもないので，この引数は無視してしまって問題ありません．

`main` の中では `printf` を呼んで，その後すぐに `return` しています．何の変哲もないプログラムですが，1 点だけ SPE ならではの話題に触れておきたいと思います．ここで呼び出している `printf` の実体は，newlib が提供している SPE 用の libc に含まれているのですが，実際に文字

列を画面に出力する処理は libc 内で行われず，PPE を呼び出して行っています．SPE は，libc 内で `printf` で出力したい文字列を LS 上に用意した後，stop-and-signal と呼ばれる命令を呼んで一旦動作を停止し，PPE に対してシグナルを送ります．libspe2 にはシグナル受信によって呼び出されるルーチン (外部関数) が用意されており，これが呼び出されると，その中で SPE が `printf` の処理を要求しているということを検出して，表示すべきデータを SPE の LS 上から読み出して標準出力などに書き出し，最後に SPE の動作を再開させるという処理を行います．このように，SPE 単体では処理が難しい関数については，PPE 側の関数を呼び出して実現するような仕組みとなっています．この辺りの機能の詳細については第 13 章にて紹介します．

`main` の第 1 引数の謎

文献 [5] を見ると，SPE の `main` の第 1 引数は PPE と SPE が共通して SPE のタスクを一意に識別するための ID であるということが書かれています．確かにコンテキストのポインタを利用すれば，必ずコンテキストごとにユニークな値となるため，SPE のタスクを PPE と SPE の両者で同じように一意に識別することが可能となります．しかしながら，ポインタがどのような番号になるのかはプログラミングを行っている段階では予想がつかないため，自身に振られた ID に応じて最初に処理を分岐させて実行する内容を切り替えるといった，一般的な並列プログラムでよく用いられる手法には利用できません．libspe2 の設計者が何を思ってここで SPE のコンテキストのポインタを利用するようにしたのか不思議でなりません．

elfspe2 と PPE プログラム

この章で紹介した Hello World の PPE プログラムは，SPE のプログラムを読み込んで SPE に実行させて終了を待つだけのことしかしておらず，メインの処理 (と言っても “Hello World” と出力するだけですが) は SPE の `main` の中で完結しています．このようなプログラムの場合，実はわざわざ PPE のプログラムを書かなくても，PS3 Linux に elfspe2 というパッケージをインストールしておくことで，以下のように SPE の実行ファイルをコマンドライン上で直接実行することができます．

```
% ./hello_spe
Hello World
```

通常，PowerPC 向けの Linux では SPE 向けの ELF ファイルの実行はサポートされていないため，本来このようなことはできないのですが，Cell SDK に付属する elfspe2 をインストールして有効にしておくと，ユーザが SPE の ELF ファイルを実行しようとした際に Linux の binfmt_misc の仕組みを用いて汎用的な PPE のプログラム (`/usr/bin/elfspe`) が呼び出されるようになり，ユーザが直接実行しようとした SPE のプログラムを，このプログラムが代わりにロードして実行してくれるようになります．

なお，elfspe2 自体は，本章で紹介した PPE のプログラムと同様に libspe2 を利用して書かれており (elfspe2 は libspe2 の付属品)，中でやっている処理も SPE プログラムを実行する部分に関しては本章で紹介した PPE プログラムとほとんど変わりありません．

第 5 章

SPE の演算機能

> 100 万パワー +100 万パワーで 200 万パワー!!
> いつもの 2 倍のジャンプがくわわって 200 万 ×2 の 400 万パワーっ!!
> そしていつもの 3 倍の回転をくわえれば 400 万 ×3 の
> バッファローマン おまえをうわまわる 1200 万パワーだーっ!!
> — ゆでたまご『キン肉マン』より —

本章では，SPE の持つ演算機能とその利用方法について紹介します．SPE はマルチメディア向けのデータ処理などの演算を専門に行うことを想定して設計されているためか，ほとんどの演算処理が SIMD 化されています．したがって，SPE の演算機能を理解し，その能力をフルに引き出すには，SIMD 演算について十分に理解しておく必要があります．

5.1　SPE の SIMD 演算

SIMD 演算とは，“Single Instruction Multiple Data” という元の言葉が示すように，単一の命令で複数のデータを同時に処理する演算です．一般的なプロセッサでの SIMD 演算は，64bit や 128bit などのビット幅の大きなレジスタに対して，それよりも幅の小さなデータを複数詰め込み，そのレジスタ全体に対して演算を行うことで，1 命令で複数のデータの演算を同時に行うことを実現します．Intel や AMD のプロセッサで利用可能な SIMD 演算には SSE や MMX などがあります．

SPE の SIMD 演算もこれらと同様で，128bit 幅のレジスタに詰め込んだ複数のデータに対して一括して演算を行う構造となっています．しかしながら，一般的なプロセッサが，汎用レジスタを用いた通常の演算 (1 命令で単一のデータを処理する演算) と，ビット幅の大きい浮動小数点数演算用のレジスタや専用の特殊なレジスタを利用した SIMD 演算の両方を提供して使い分けるようになっているのに対し，SPE ではこのような使い分けはありません．SPE では，すべての汎用レジスタに対して SIMD 演算を適用できる上に，ほとんどの演算処理が SIMD 演算となっているのです．

SPE の SIMD 演算は，インラインアセンブラなどを用いてアセンブリ言語による記述を C 言語のソースコード中に埋め込んだりしなくても，普通に C 言語から intrinsic と呼ばれる組み込み関数を呼ぶだけで利用することができるようになっています．また，前述の通り，SPE ではほとんどの演算処理が SIMD 演算となっているため，C 言語のソースコード上で単純に `int` 型の整数同士の足し算を記述した場合でも，SIMD 演算を用いた命令列が生成されます (その場合，

演算結果の 3/4 は無意味な値となる).

なお，本章で紹介する組み込み関数などを利用するには，SPE のプログラムで `spu_intrinsics.h` というヘッダファイルを `include` しておく必要があります．自分で実際に動かして確かめる場合は適宜ソースの先頭に記述してください．

5.2　バイトの並びとビットの番号

SPE 固有の演算の話に入る前に，ビッグエンディアン環境でのバイトの並びやビットへの番号の振り方を確認しておきます．Cell/B.E. 関連の各種ドキュメントで用いられている表記を理解する上でこの知識は必須ですので，何を今更と思われる方も，是非一度目を通しておくとよいでしょう．

5.2.1　ビッグエンディアンのバイト並び

最初にビッグエンディアンの基本について確認します．ビッグエンディアンは，2Byte や 4Byte などの複数バイトで構成されるデータをメモリに格納する際に，**上の桁から 1Byte ずつ順番に，アドレスの小さい方から格納していく**方式です．ここで，メモリの 0x0014 番地に 0x01234567 という 4Byte のデータを格納した場合のデータの配置を考えます．C 言語で書くとリスト 5.1 のようになります．このプログラムを実行した場合，ビッグエンディアンな環境では，メモリの 0x0014 番地に 0x01 が入り，0x0015 番地に 0x23 が入ります．

リスト 5.1　アドレスを明示したメモリへの格納の例

```
*(unsigned *)0x0014 = 0x01234567;
```

図 5.1 にこの様子を示します．

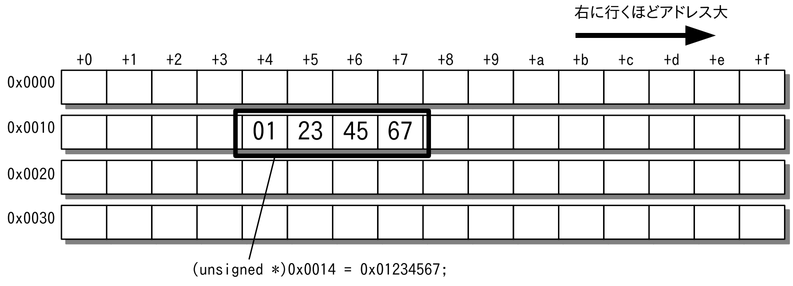

図 5.1　ビッグエンディアン環境でのメモリ上のデータの並び

ビッグエンディアンでデータがメモリ上にどう並ぶかを考える場合は，図 5.1 に示したような，左から右にアドレスが増えるメモリの並びを描いておくとよいでしょう．こうしておくことで，人間が普段扱っている桁の並びと同じ順番で図にデータが現れるようになるため，混乱が減ります[*1]．図があれば，メモリの 0x0014 番地に 0x01 が入っており，0x0015 番地に 0x23 が入っていることが一目で理解できることでしょう．

[*1] 逆にリトルエンディアンの場合は右から左にかけてアドレスが増えるような図を描いて考えるとわかりやすいです．

次に，先ほどデータを格納したアドレス (0x0014 番地) を，他の型のポインタにキャストして読み出すことを考えてみます．たとえばリスト 5.2 のように，unsigned short *型にキャストして中身を読んだとしましょう．

リスト 5.2　キャストを伴ったメモリからの読み出しの例

```
unsigned short x;
x = *(unsigned short *)0x0014;
```

一瞬 0x0123 になるのか，それとも 0x4567 になるのかで混乱しそうですが，先ほどの図を用いて簡単に判断する方法があります．0x0014 番地から 2Byte 分ということになるので，図 5.2 に示すように 0x0014 番地から 2 マス分を枠で囲んでみてください．枠で囲まれた中に現れるデータが読み出されるデータになります．よって，この例では 0x0123 が読み出されることになります．

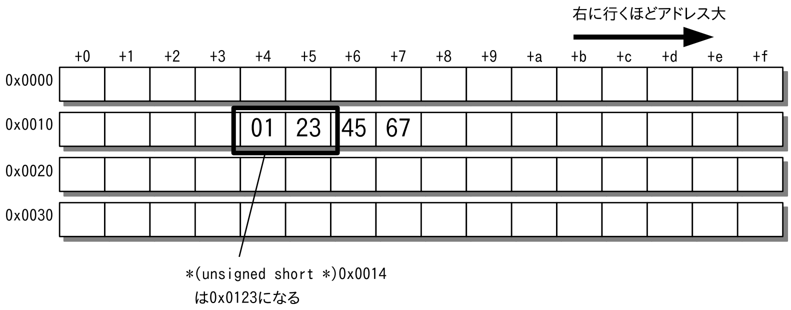

図 5.2　ビッグエンディアン環境でポインタをキャストしてデータを読んだ場合

同様に unsigned char *型にキャストして中身を読んだ場合は，0x0014 番地から 1 マス分だけを枠で囲みます．枠で囲まれた中にあるデータが読み出されるデータとなるので，unsigned char *型にキャストして読み出した場合は 0x01 が読み出されることがわかります．

ビッグエンディアンのデータ配置でよく混乱するという方でも，図を見ると当たり前過ぎる話と感じたのではないでしょうか．もしそうでしたら，これを機に図の描き方を覚えてしまいましょう．脳内でビッグエンディアンでのデータのキャストの結果をイメージするのが苦手でも，このように図に描いて考えると確実で間違いも減ります．

5.2.2　Cell/B.E. でのビットの番号

次に，Cell/B.E. でのビットへの番号の割り振り方についても確認しておきます．ビッグエンディアンだからなのか，それとも IBM 流だからなのかわかりませんが，Cell/B.E. 関連のドキュメントでは “データの何ビット目” というのを表現する際に用いるビットの番号がちょっと普通と違うので注意が必要です．Cell/B.E. のドキュメント類では，データの一番上の桁のビット (人間が 2 進数で書き直したときに一番左に来るビット，いわゆる Most Significant Bit (MSB)) がビット 0 で，右に行くにしたがって 1，2，⋯ と増えていく番号付けがされています．

言葉だけではわかりにくいので，これも図にしてみましょう．図 5.3 に 32bit，16bit および 8bit のデータのビットに対する番号の振り方を示します．図の 1 つのマスが 1 つのビットに対応するものと思ってください．考えやすいように，それぞれ 0x01234567，0x89ab，0xcd が格納

されている状態のビット並びを示しています．

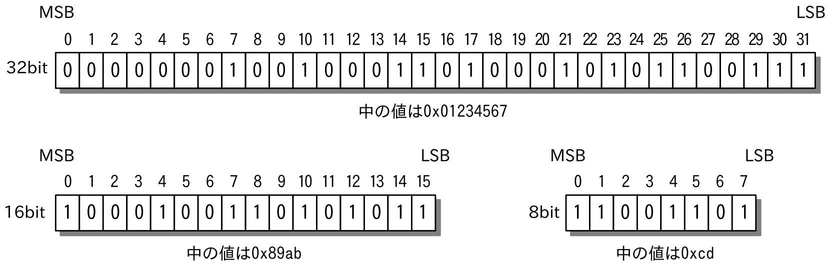

図 5.3　ビットの番号の振り方の例

図 5.3 のように，ビット幅に関わらず，いずれの幅のデータにおいてもビット列の左端のビット (MSB) に番号 0 を振り，そこから右へいくに従ってビット番号が増えていきます．そのため，一番右端のビット (Least Significant Bit (LSB)) に振られる番号は，ビット幅によって異なることになります．64bit や 128bit のビット列についても同様です．

Cell/B.E. のドキュメントではすべてこのような番号の振り方となっているため，レジスタのビットの並びを確認する際などは注意が必要です．筆者はこの表記が好きではないのですが，本書でも他の Cell/B.E. 関連のドキュメントに合わせてこの表記でビットの位置を表現することにします．

5.3　ベクタとスカラ

5.1 で述べたように，SPE の演算処理のほとんどは SIMD 演算となっており，128bit 幅の汎用レジスタに詰め込んだ (パックした) 複数のデータに対して 1 命令で一斉に同じ演算処理を適用することができるようになっています．その際，SPE は “32bit 幅の `float` 型のデータが 4 個” や “8bit 幅の `char` 型のデータが 16 個” のように，汎用レジスタに詰め込まれているデータがすべて同じ型であるとみなして演算を行います．このような 128bit 幅の中に同じ型のデータをパックしたものを**ベクタ**と呼びます．また，これに対してベクタの 1 要素 (パックされるもの) を構成する `int` 型や `float` 型などの型のデータを**スカラ**と呼びます．

SPE の SIMD 演算では 10 種類のベクタを扱うことができます．いずれのベクタも 128bit 幅ですが，パックされているデータの型が各々異なります．表 5.1 に SPE で利用できる 10 種類のベクタの型を示します．これら 10 種類のベクタは SPE のコンパイラで標準的に利用可能なデータ型としてサポートされており，プログラム中では “ベクタのデータ型名” の列に示した名称を用いて型を扱うことができます．また，“要素の型” と “個数” の列には，その行のベクタの型が，どのようなスカラを何個パックしたものなのかが示されています．いずれの型も 128bit ぴったりに収まるようなスカラの型と個数の組み合わせになっていることがわかります．なお，表の “省略形” の列には各ベクタの型の別名が示してあります．省略形も通常のベクタの型名と同じようにコンパイラによってサポートされています．プログラム中で “`vector unsigned long long`” などといちいち書くと非常に長くなってしまうので，本書では省略形を用います．

表 5.1 SPE で利用可能なベクタの型

ベクタのデータ型名	要素の型	個数	省略形
vector unsigned char	unsigned char 型	16	vec_uchar16
vector signed char	signed char 型	16	vec_char16
vector unsigned short	unsigned short 型	8	vec_ushort8
vector signed short	short 型	8	vec_short8
vector unsigned int	unsigned int 型	4	vec_uint4
vector signed int	int 型	4	vec_int4
vector unsigned long long	unsigned long long 型	2	vec_ullong2
vector signed long long	long long 型	2	vec_llong2
vector float	float 型	4	vec_float4
vector double	double 型	2	vec_double2

5.3.1 ベクタの扱い

SPE のプログラムでは，表 5.1 で示したベクタの型を用いてベクタ型の変数を宣言することができます (ベクタ変数)．ベクタ変数はコンパイラによって汎用レジスタに直に割り付けられます．C 言語で書かれたプログラムの中で，SIMD 演算などの SPE の演算処理を明示的に利用する場合，ほとんどの場合においてベクタ変数が必要となるので，是非ここでその扱い方について理解してください．

まず，ベクタ変数は，sizeof 演算子の適用や同じ型同士の変数間での代入記号による値のコピー，配列としての確保，ポインタ演算などが可能です[*2]．構造体などに適用できる処理がほぼそのまま利用可能であると考えれば理解しやすいでしょう[*3]．

また，通常のスカラ変数や構造体などのポインタをベクタ型のポインタにキャストした上でベクタ変数に読み込むといったことも可能です．配列の値をベクタ変数に読み込む例をリスト 5.3 に示します．

リスト 5.3 ベクタへの配列の値のロードの例

```
unsigned d[] __attribute__((aligned(16))) = {1, 2, 3, 4, 5, 6, 7, 8};
...
    vec_uint4 v1, *vp1;
    vp1 = (vec_uint4 *)d;
    v1 = *vp1;
```

リスト 5.3 では，配列 d に “__attribute__((aligned(16)))” というディレクティブが付随しているため，この配列は 16Byte 境界にアラインされた (配列の先頭のアドレスが 16 の倍数となった) 状態で確保されます[*4]．v1 は，要素が unsigned 型のベクタ変数で，vp1 は

*2 実は spu-gcc では “+” や “|” などの通常の演算子を用いてベクタ変数同士の間で SIMD 演算をすることもできてしまうのですが，後々命令の並びの最適化などがしにくくなるので，本書ではできないものとして話を進めます．

*3 ただしベクタは実際には構造体ではないので，ドットで区切ってメンバ名を指定して個々の要素を参照するといったことはできません．

*4 __attribute__((aligned(n))) の詳細については 10.1.5 で述べます．

vec_uint4 型のベクタ変数のポインタになります．リストでは，まず unsigned *型である d を vec_uint4 *型にキャストして vp1 に格納しています．vp1 自体の値は d の値と同じですが，これに “*” をつけると，vp1 の示すアドレス (すなわち d の先頭) から vec_uint4 型のデータを 1 個分取り出すことになります．vec_uint4 型は 128bit 幅なので，v1 には d[0] から d[3] までのデータが格納されることになります．

一方，リスト 5.4 のように 16Byte 境界にアラインしていないデータを直接ベクタ変数に読み出すことはできません．

リスト 5.4 ベクタへの配列の値のロードがうまくいかない例

```
unsigned d[] __attribute__((aligned(16))) = {1, 2, 3, 4, 5, 6, 7, 8};
...
    vec_uint4 v2, *vp2;
    vp2 = (vec_uint4 *)(d + 1);
    v2 = *vp2;
```

この例では，d の先頭ではなく d[1] のアドレスを vec_uint4 *型にキャストした上で vp2 に格納しています．したがって，v2 に読み込もうとしているデータの開始アドレスは 16 の倍数＋4 となり，16Byte 境界にアラインしていないことになります．このような記述を行った場合，v2 には d[1] から d[4] までのデータではなく，リスト 5.3 と同じように d[0] から d[3] までのデータが格納されます．これは，ベクタに対して LS から値をロードする際に，LS 側のアドレスの下位 4bit が無視され，強制的に 16Byte 境界にアラインしたところから読み込まれるようになっているためです．

なお，このように 16Byte 境界からずれたところに置かれたデータを単一のベクタ変数にロードする場合，通常は 2 回ロードを行った上で結果を混ぜるといったことを行います．詳しい方法については後ほど 6.3.4 にて紹介しますが，とにかくここでは，ベクタ変数には 16Byte 境界にアラインした領域からしかデータを読み込むことができないということを覚えておきましょう．

5.3.2 ベクタのリテラル

プログラム中でベクタ変数に初期値を代入したい場合などは，リテラル表現でベクタの値そのものを記述する必要があります．ベクタの値そのもののリテラル表現をベクタリテラルと呼びます．ベクタリテラルは “(ベクタ型名){要素 0，要素 1，...}” のように記述します．

リスト 5.5 にベクタリテラルの使用例を示します．この例の v1 と v2 は，ともに vec_ushort8 型のベクタで，v1 には宣言と同時にベクタリテラルを用いて初期値が設定されています．また，v2 には宣言の後で別途ベクタリテラルで指定した値が代入されています．ベクタリテラルを利用することで，通常のスカラ変数に対する定数の代入のように，ベクタ変数に値を直に格納することができることを覚えておきましょう．

リスト 5.5 ベクタリテラルの使い方の例

```
    vec_ushort8 v1 = (vec_ushort8){0, 1, 2, 3, 4, 5, 6, 7}, v2;
    v2 = (vec_ushort8){8, 9, 10, 11, 12, 13, 14, 15};
```

5.3.3　スロットとプリファードスロット

ベクタ変数は，演算の際に 128bit 幅のレジスタに割り付けられますが，その際，ベクタを構成する要素が格納される個々の領域 (何ビット目から何ビット目までというビット範囲) をスロットと呼びます．スロットには幅と位置に応じた名称がついています．スロットの幅と位置に応じた名称を図 5.4 に示します．

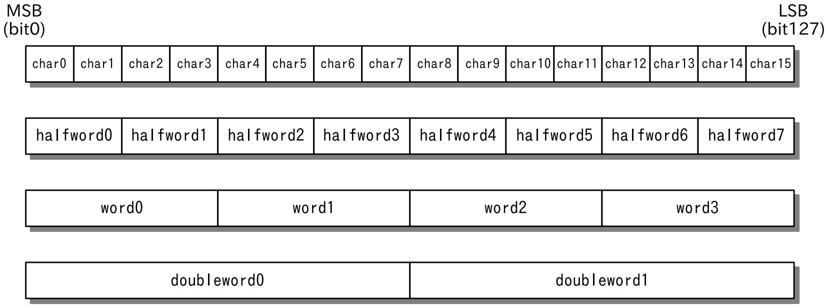

図 5.4　幅と位置に応じたレジスタのスロットの名称

Cell/B.E. 関連の各種ドキュメントを見ると，“偶数番目の要素” や “奇数番目の要素” といった表現が頻繁に現れますが，これらは word1 や char8 などの，スロットの名称の末尾についている番号に対応します．スロットの名称についている番号が 0 オリジンである点に注意してください．MSB に最も近いスロットに含まれるのは偶数番目の要素となります．

また，5.1 で述べたように，SPE はスカラの変数やリテラルに対して演算を行う場合でも，128bit 幅の汎用レジスタを利用して SIMD 演算を行います．このような場合，SPE は汎用レジスタの特定スロットにのみ演算したいデータを入れておき，他のスロットの中身は不定なままで演算を行って，結果が格納されたレジスタの同じ位置のスロットのみを参照することで結果を得ます．スカラの演算で利用するスロットは，スカラの型 (のビット幅) によって表 5.2 のように決まっています．

表 5.2　スカラの型とレジスタ上で利用するスロット

スカラの型	ビット幅	利用するスロット
`char`，`unsigned char`	8	char3
`short`，`unsigned short`	16	halfword1
`int`，`unsigned`，`float`	32	word0
`long long`，`unsigned long long`，`doulbe`	64	doubleword0

たとえば `int` 型の変数はレジスタの word0 を利用し，`char` 型の変数はレジスタの char3 を利用します．スカラが利用するこのような特定のスロットを “**プリファードスロット**” と呼びます．

図 5.5 にプリファードスロットの位置を示します．図の黒い枠で囲われた部分がプリファードスロットになります．必ずしも 128bit のレジスタの一番左側のスロットがプリファードスロットとなるわけではないという点に注意してください．

後で詳しく述べますが，プリファードスロットはベクタとスカラの間での変換を効率的に行う

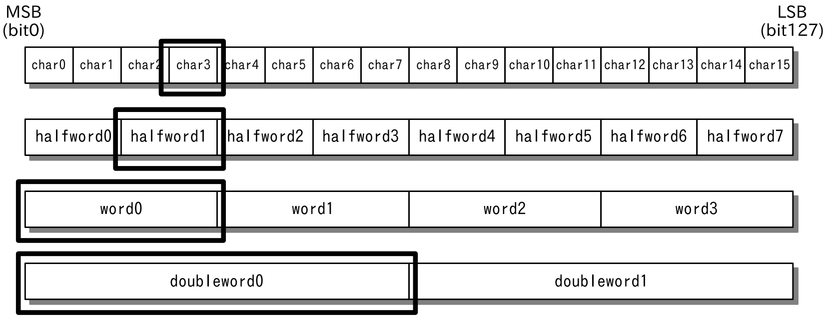

図 5.5　プリファードスロットの位置

場合などに考慮する必要があります．是非理解しておきましょう．

5.4　組み込み関数による SPE の演算処理の利用

ここからは，組み込み関数と呼ばれる仕組みを用いた，SPE の持つ演算処理の利用について説明していきます．

SPE の演算処理は，基本的に SPE の命令 1 つ 1 つにそのまま対応します．C 言語のプログラム中で，SPE の持つ演算処理やその他の命令を呼び出して利用するには，インラインアセンブラを用いて直接 SPE の命令を記述するか，あるいは各種命令を狙って呼び出すのに使えるコンパイラの組み込み関数 (intrinsic) を利用する必要があります[*5]．前者を使った方がより高度な感じがして格好良いのですが，本書では後者を用います．

組み込み関数と一口に言っても，後で紹介するように命令に一対一対応するものもあれば，もっと抽象化されたものもあり，SPE の命令との対応の仕方も様々なのですが，基本的に命令のソースオペランド (演算対象となるレジスタなど) が関数の引数の形で表され，デスティネーションオペランド (演算結果が格納されるレジスタ) が関数の戻り値として表されます[*6]．イメージが湧かないかもしれないので，後ほど具体例を紹介します．

5.4.1　spu-gcc の 3 種類の組み込み関数

spu-gcc で用意されている組み込み関数には以下の 3 種類があります．

- 型固有組み込み関数 (Specific Intrinsics)

  型固有組み込み関数[*7]は，SPE の命令に一対一対応する組み込み関数です．この関数を呼ぶことで，ほぼすべての SPE の命令を C 言語のソースコード上で関数の形で呼び出すこと

[*5] もちろん，スカラ変数に対する演算処理は通常の “+” や “*” などの演算子を用いることで可能ですが，SPE の演算処理にはビットのローテーションなどの C 言語の演算子では表現できないようなものも多数存在するため，すべての演算命令を演算子のみで自在に呼び出すことは難しいでしょう．

[*6] SPE の命令にはソースオペランドがデスティネーションオペランドを兼ねるタイプの命令がないため，このような形で統一することができています．

[*7] 文献 [2] などでは “個別組み込み関数” という名称が使われていますが，名前から機能がイメージしにくいので，本書では “型固有組み込み関数” とします．ちなみに個別組み込み関数という邦訳は FORTRAN の世界などでは普通に使われているようです．

ができます.

プロセッサの演算命令は，同種の演算であっても，処理するデータの種類 (型) や対象 (レジスタなのか即値 (命令に埋め込まれた定数) なのか) などの違いによって複数用意されているのが一般的です．SPE でもこれに関しては同様で，たとえばレジスタ同士の加算を行うにしても，レジスタに 16bit 幅の整数が 8 個のパックされている場合と 32bit 幅の整数が 4 個のパックされている場合とで異なる処理を行う必要があるため，異なる命令が用意されています．このような命令に対して一対一対応するように組み込み関数を用意した場合，それぞれが扱うデータの型や対象に固有の関数となることから，“型固有” 組み込み関数と呼んでいます.

SPE の型固有組み込み関数の関数名は “`si_`” で始まり，その後ろに SPE の命令名を付加したものとなります．たとえばレジスタに 16bit 幅の整数が 8 要素パックされているとみなして，2 つのレジスタの各要素同士を足し算する “`ah`” という SPE の命令に対応する型固有組み込み関数は “`si_ah`” という関数名になります．また，`ah` のアセンブリ言語のフォーマットは “`ah　rt, ra, rb`” であり，3 つのオペランドを取る命令となっていますが，このうち `ra` と `rb` がソースオペランドで，`rt` がデスティネーションオペランドとなることから，対応する型固有組み込み関数は “`d = si_ah(a, b)`” のように，引数を 2 つ取って値を返す型となります.

後述する型総称組み込み関数の方が便利なので，本書では型固有組み込み関数はほとんど扱いませんが，ロード命令やストア命令，NOP 系の命令などは型固有組み込み関数でしか狙って呼び出せないので，覚えておくといざというときに役に立つかもしれません．なお，分岐や割り込みなどに関連する命令については，一部の例外を除いて型固有組み込み関数でも呼び出すことができません (対応する型固有組み込み関数が用意されていない)．型固有組み込み関数で呼び出すことのできない命令の一覧や，型固有組み込み関数を用いて実際に呼び出すことのできる個別の命令の挙動などについて詳しく知りたい場合は，文献 [2] および文献 [3] を参照してください.

- 型総称組み込み関数 (Generic Intrinsics)

型総称組み込み関数[*8]は，関数に引数として与えたデータの型に応じて異なる SPE の命令 (あるいは命令列) に置き換わる組み込み関数です．型固有組み込み関数が命令に一対一対応して用意されており，扱うデータの型などに応じて関数を呼び分けなければならなかったのに対し，型総称組み込み関数は “ベクタの要素同士を足し算する” や “ベクタの各要素を左シフトする” といった機能ごとに用意されており，単一の関数で様々な型の引数を取ることができるようになっています (一部例外的に引数の型が 1 種類しか取れないものもあり).

たとえば，SPE には，2 つのレジスタをベクタとみなし，それらの要素同士を足し合わせる命令として，`a` (`vec_int4`・`vec_uint4` 用)，`ah` (`vec_short8`・`vec_ushort8` 用)，`fa` (`vec_float4` 用) および `dfa` (`vec_double2` 用) の 4 種類が用意されています．また，

*8 文献 [2] などでは “総称組み込み関数” という名称が使われていますが，こちらも名前から機能がイメージしにくいので，本書では “型総称組み込み関数” とします.

ベクタの各要素に対して即値を足す命令についても，ai (vec_int4・vec_uint4 用) と ahi (vec_short8・vec_ushort8 用) の 2 種類が用意されています．これらを型固有組み込み関数で利用しようと思うと，足し合わせるベクタの型に応じて si_a，si_ah，si_fa，si_dfa，si_ai，si_ahi の 6 種類の型固有組み込み関数を呼び分けなければなりません．これに対し，型総称組み込み関数を用いた場合，引数のベクタの型によらず spu_add という関数を呼ぶだけでコンパイラが適切な SPE の命令の呼び出しに変換してくれます*9．オブジェクト指向などで出てくるポリモーフィズムの概念と同様というとわかりやすいかもしれません．C 言語らしくないので抵抗を覚えるかもしれませんが，本書ではこの型総称組み込み関数を中心に用います．本章で紹介するのも，この型総称組み込み関数 (の中で SIMD 演算と関係するもの) になります．

なお，型総称組み込み関数の引数と戻り値も，型固有組み込み関数と同様に，変換される命令のソースオペランドとデスティネーションオペランドに対応します．したがって，機能的に似ていてもオペランドの個数が異なる命令は，異なる型総称組み込み関数として定義されています．

- 複合組み込み関数 (Composite Intrinsics)

  複合組み込み関数は，関数の呼び出しが複数の型総称組み込み関数や型固有組み込み関数の呼び出しに置き換わる組み込み関数です．spu-gcc などで利用可能な複合組み込み関数には DMA 転送関連のものしかなく，本章で紹介する演算処理を呼び出す上では必要ありません．第 10 章にて DMA 転送について説明する際にあらためて紹介します．

5.4.2　組み込み関数の使用例

ここまでの説明で，spu-gcc ではどのような種類の組み込み関数が利用可能かということを紹介しましたが，これだけでは具体的なイメージがなかなかつかみにくいかと思うので，ここで型総称組み込み関数を利用した簡単な記述の例を示します．リスト 5.6 は，型総称組み込み関数を使ってベクタ変数同士の足し算を行う記述になります．最初にベクタ変数 va に d[0] から d[3] までを読み込み，ベクタ変数 vb に d[4] から d[7] までを読み込んだ上で，型総称組み込み関数 spu_add に引数として渡して，結果を vd に代入しています．したがって，この例では unsigned 型の変数同士の足し算を 4 組まとめて行っていることになります．このように，組み込み関数は通常の関数とまったく同じ形で利用することができます．

リスト 5.6　型総称組み込み関数を使ったベクタ変数同士の足し算

```
unsigned d[] __attribute__((aligned(16))) = {1, 2, 3, 4, 5, 6, 7, 8};
...
    vec_uint4 va, vb, vd, *vp;
    vp = (vec_uint4 *)d;
    va = *vp;
    vb = *(vp + 1);
    vd = spu_add(va, vb);
```

*9 もちろん，変換できるのは SPE が実際にハードウェアで提供している命令に過ぎないので，SPE の命令が対応していない型が引数に与えられた場合はコンパイルエラーとなります．

5.5 演算処理向けの型総称組み込み関数

ここからは，SPE の演算処理の呼び出しに用いる型総称組み込み関数を順番に紹介していきます．ページ数の都合上，すべての型総称組み込み関数の振る舞いを網羅することは困難なので，個々の説明は押さえておくべき重要なポイントに絞って行っています．各型総称組み込み関数に対応する SPU の命令や，その命令の詳しい振る舞いなどについては，別途文献 [2] や文献 [3] などを参照してください．

5.5.1 ベクタとスカラの変換

いきなり出端をくじくようで申し訳ないのですが，演算のための型総称組み込み関数を紹介する前に，ベクタとスカラの間の変換を行う型総称組み込み関数を紹介しておきます．これがあると，ベクタを柔軟に扱えるようになるため，この先紹介する様々な型総称組み込み関数を実際に呼び出して試す際に何かと役立ちます．

spu_extract：ベクタの要素の抽出

```
d = spu_extract(a, e);
```

spu_extract は，ベクタの中から引数で指定した要素を取り出してスカラで返します．したがって，戻り値は抽出元のベクタの各要素と同じ型のスカラとなります．上記のフォーマットの場合，ベクタ a の e 番目の要素の値がスカラ変数 d に格納されます．たとえば，a が vec_short8 型で e が 7 だとすると，a の halfword7 の値が d に格納されます．a は 10 種類すべてのベクタに対応します．また，e は int 型の変数もしくはリテラルに対応します．細かい話になりますが，何番目の要素を取り出すかを指定する e は，a の各要素のサイズに応じて下位ビットの一部のみが利用されます．たとえば a が vec_uchar16 の場合は e の下位 4bit のみが利用され，a が vec_double2 の場合は e の下位 1bit のみが利用されます．

spu_extract は特別な SPU の命令で実現されているわけではなく，内部的には 5.5.10 で紹介するシフトやローテートの機能を駆使して目的の値をプリファードスロットの位置に持ってくるということが行われます．ただし，取り出す対象となる要素のスロットがプリファードスロットであることが確実な場合 (たとえば a が vec_char16 型のベクタで，e がリテラルの 3 の場合など) に限り，この型総称組み込み関数はコンパイル時に何の命令にも置き換わることなく消滅します．これは，スカラ変数は，レジスタのプリファードスロットにさえ正しいデータが格納されてさえいれば，他のスロットには何が入っていようと構わない扱いとなっているためです．

spu_extract は，リスト 5.7 のように用いることで，ベクタの要素を出力する用途に使えます．本章で紹介する型総称組み込み関数の振る舞いを確認する際や，実際のプログラミングの過程でデバッグを行う際などに便利です．

なお，spu_extract は比較的処理に時間のかかる命令に変換されてしまうため，プリファードスロットを抽出するのが確実な場合は別として，デバッグなどの用途以外では極力利用を避けることをおすすめします．

リスト 5.7　spu_extract を利用してベクタの各要素を printf で出力する例

```
int i;
vec_uint4 vu1;
...
for (i = 0; i < 4; i++)
    printf("0x%02x ", spu_extract(vu1, i));
printf("\n");
```

spu_insert : ベクタの要素の置き換え

```
d = spu_insert(a, b, e);
```

spu_insert は，ベクタの 1 要素を指定したスカラの値で置き換えたベクタを返します．上記のフォーマットの場合，ベクタ b の e 番目の要素の値を変数 a の中身で置き換えて，ベクタ d に格納します．b は 10 種類のすべてのベクタに対応し，a はスカラの変数に，e は int 型の変数およびリテラルに対応します．なお，当然ですが，a の型は b の要素の型と同じでなければなりません．また，引数の並び順がややこしいので注意してください．

何番目の要素を置き換えるかを指定する e は int 型ですが，b の各要素のサイズに応じて下位ビットの一部のみが利用されます．たとえば b が vec_uchar16 の場合は e の下位 4bit のみが利用され，b が vec_double2 の場合は e の下位 1bit のみが利用されます．

spu_insert も，spu_extract と同じように比較的処理に時間のかかる命令に変換されるため，コアループの中などでは使わない方がよいでしょう．

spu_splats : スカラを複製してベクタに変換

```
d = spu_splats(a);
```

spu_splats は，引数として指定したスカラの変数やリテラルを複数並べてベクタにしたものを返します．関数の最初の方でベクタを初期化する際などによく用います．上記のフォーマットの場合，a の型に応じて a の値を 2 個から 16 個並べたものがベクタとして d に格納されます．a は SPE でベクタの要素として扱える 10 種類すべての型のスカラの変数もしくはリテラルに対応します．たとえば a の型が unsigned short 型で値が 0x1234 だった場合，d の全 8 要素に 0x1234 が格納されます．

spu_promote : スカラをそのままベクタに変換

```
d = spu_promote(a, e);
```

spu_promote は，引数で与えたスカラの変数の値を指定した位置に格納したベクタを返します．上記のフォーマットの場合，スカラの変数 a の値が e で指定されるスロットに格納されたベクタを新たに生成して d に格納します．d の型は a の型に応じて決まり，a は SPE でベクタの要素として扱える 10 種類すべての型に対応します．たとえば a が int 型であれば vec_int4 型のベクタが返ることになります．また，スロットを指定する e は int 型の変数かリテラルに対応し，spu_insert などと同様に a のサイズに応じて下位ビットの一部のみが利用されます．なお，d に格納されるビット列において，e で指定したスロット以外のスロットに位置する要素の

値は適当な値となります (0 が入るとは限らない).

spu_promote についても，spu_extract などと同じように比較的処理に時間のかかる命令に変換されるため，使用する際は注意が必要です．また，spu_extract と同様に，e がリテラルで，ベクタに変換した際に a がちょうどプリファードスロットに収まるような引数の組み合わせとなった場合，変換処理が不要となるため，特別な命令は生成されません．

5.5.2　加減算

以下では加減算に関わる SIMD 演算を行うための型総称組み込み関数を紹介します．

spu_add：ベクタの要素同士の足し算

```
d = spu_add(a, b);
```

spu_add は，ベクタの要素同士の足し算もしくはベクタとスカラの足し算を行います．上記のフォーマットの場合，a の各要素と b の各要素 (b がスカラの変数やリテラルの場合は b を複製したもの[*10]の各要素) を足し算した結果が d の各要素に格納されます．なお，a や b には要素が 8bit 幅の整数のベクタおよび要素が 64bit 幅の整数のベクタは指定できません．vec_llong2 型や vec_ullong2 型のベクタの要素同士の足し算は，後述の桁上がりを含む足し算を用いて実現します．

spu_genc・spu_addx・spu_gencx：ベクタの要素同士の足し算 (桁上がりつき)

```
d = spu_genc(a, b);
d = spu_addx(a, b, c);
d = spu_gencx(a, b, c);
```

これら 3 つの型総称組み込み関数は，桁上がり (キャリー) の関係する加算を行います．いずれも vec_int4 型および vec_uint4 型のベクタのみを引数として取ることができます．

spu_genc は，spu_add と似ていますが，各要素同士の足し算の結果そのものではなく，**足し算によって桁上がりが発生するかどうか**の情報をベクタとして返します．上記のフォーマットの場合，a の各要素と b の各要素を足した結果，桁上がりが発生するならば d の対応する要素に 1 を格納し，そうでないならば 0 を格納します．たとえば a の word2 の値が 0xffffffff で，b の word2 の値が 0x00000001 だった場合，足すと桁上がりが発生するので，d の word2 に 1 が格納されます．この spu_genc で求めた桁上がりの有無は，spu_addx や spu_gencx で使うことができます．

spu_addx は，2 つのベクタの要素と 1 つのベクタの要素の LSB の足し算を行います．上記のフォーマットの場合，a の各要素と b の各要素の加算を行い，さらにその結果に c の各要素の LSB の加算を行って，結果を d の各要素に代入します．c に spu_genc の結果を取ることで，桁上がりを含んだ足し算を行うことができます．したがって，spu_addx を spu_genc の結果と組み合わせることで，64bit 幅の整数同士の加算を実現することができます．

[*10] b を a の要素数分たくさん並べてベクタ化したもの．

spu_genc と spu_addx を組み合わせて vec_llong2 型のベクタの要素同士の足し算を行う関数の例をリスト 5.8 に示します．ここでは 2 つの vec_llong2 型のベクタを vec_int4 型のベクタとみなして足し算を行っています．vec_llong2 型のベクタの各要素の下位 32bit 同士の足し算で桁上がりが発生した場合，それを上位 32bit 同士の足し算に反映させる必要があるため，先に spu_genc で桁上がりの発生を調べ，spu_shuffle を用いて word1 および word3 に格納されているそれらの結果を word0 および word2 に移しつつ word1 および word3 に 0 を入れ，最後に spu_addx で足し合わせています．桁上がりの情報を動かすのに使っている spu_shuffle については 5.5.7 で紹介します．

リスト 5.8　spu_genc と spu_addx を利用した vec_llong2 型のベクタ同士の足し算

```
#define CARRY_PATTERN                                                  \
    (vec_uchar16){0x80, 0x80, 0x80,  7, 0x80, 0x80, 0x80, 0x80,  \
                  0x80, 0x80, 0x80, 15, 0x80, 0x80, 0x80, 0x80}

vec_llong2
add_llong2(vec_llong2 a, vec_llong2 b)
{
    vec_int4 c;
    // spu_gencで扱えるようにvec_int4型にキャストする
    c = spu_genc((vec_int4)a, (vec_int4)b);
    // word1とword3の結果をword0とword2に移し，word1とword3はゼロ埋め
    c = spu_shuffle(c, c, CARRY_PATTERN);
    return (vec_llong2)spu_addx((vec_int4)a, (vec_int4)b, c);
}
```

また，spu_addx によって桁上がりが発生するかどうかについては，spu_gencx を利用することで知ることができます．spu_gencx は，2 つのベクタの要素と 1 つのベクタの要素の LSB の足し算によって桁上がりが発生するかどうかについての情報をベクタとして返します．上記のフォーマットの場合，a の各要素と b の各要素を足し，さらにそれに c の各要素の LSB を足し合わせた場合に桁上がりが発生するならば d の対応する要素に 1 を格納し，そうでないならば 0 を格納します．データの配置を工夫しておけば，これを用いることで 96bit 幅や 128bit 幅の整数の足し算なども可能となるでしょう．

spu_sub：ベクタの要素同士の引き算

```
d = spu_sub(a, b);
```

spu_sub は，ベクタの要素同士の引き算もしくはスカラからベクタの引き算を行います．上記のフォーマットの場合，a の各要素 (a がスカラの変数やリテラルの場合は a を複製したものの各要素) から b の各要素を引いた結果が d の各要素に格納されます．spu_add と同様に，a や b には要素が 8bit 幅の整数のベクタおよび 64bit 幅の整数のベクタは指定できません．vec_llong2 型や vec_ullong2 型のベクタの要素同士の引き算は，後述の桁借りを含む演算を用いて実現します．

なお，**スカラを指定できるのは a に限られるため，スカラを複製したものからベクタを引くことはできても，その逆はできません．**ベクタからスカラを複製したものを引きたい場合は，spu_add の第 2 引数にスカラの符号を反転させたものを与えるなどします．

spu_genb・spu_subx・spu_genbx：ベクタの要素同士の引き算 (桁借りつき)

```
d = spu_genb(a, b);
d = spu_subx(a, b, c);
d = spu_genbx(a, b, c);
```

これら 3 つの型総称組み込み関数は，桁借り (ボロー) の関係する減算を行います．いずれも vec_int4 型および vec_uint4 型のベクタのみを引数として取ることができます．

spu_genb は，spu_sub と似ていますが，各要素同士の引き算の結果ではなく，引き算の結果で桁借りが発生するかどうかの情報をベクタとして返します．上記のフォーマットの場合，a の各要素から b の各要素を引いた結果，**桁借りが発生するならば d の対応する要素に 0 を格納し，そうでないならば 1 を格納します．桁借りが発生した場合に 0 が格納される**という点に気をつけてください．たとえば a の word0 が 1 で b の word0 が 2 の場合，引き算した際に桁借りが発生するので，d の word0 には 0 が格納されます．この spu_genb で求めた桁借りの有無は，spu_subx や spu_genbx で使うことができます．

spu_subx は，2 つのベクタの要素間で引き算を行い，さらにそこから別のベクタの要素の LSB を反転した値 (LSB が 0 だったら 1，1 だったら 0) を引きます．上記のフォーマットの場合，a の各要素から b の各要素を引き，c の対応する要素の LSB が 0 だった場合はさらにそこから 1 を引いた上で，結果を d の対応する要素に代入します．c に spu_genb の結果を取ることで，桁借りを含んだ引き算を行うことができます．したがって，spu_subx を spu_genb の結果と組み合わせることで，64bit 幅の整数同士の減算を実現することができます．

spu_genb と spu_subx を組み合わせて vec_llong2 型のベクタの要素同士の引き算を行う例をリスト 5.9 に示します．ここでは 2 つの vec_llong2 型のベクタを vec_int4 型のベクタとみなして引き算を行っています．vec_llong2 型のベクタの各要素の下位 32bit 同士の引き算を行って桁借りが発生した場合，それを上位 32bit 同士の引き算にさらに反映させる必要があるため，先に spu_genb で桁借りの発生を調べ，spu_shuffle を用いて word1 および word3 に格納されているそれらの結果を word0 および word2 に移しつつ word1 および word3 の LSB に 1 が入るようにして，最後に桁借り情報も含めて spu_subx で実際の引き算を行います．桁借り発生の情報を別のスロットに動かすのに使っている spu_shuffle については 5.5.7 で紹介します．

リスト 5.9　spu_genb と spu_subx を利用した vec_llong2 型のベクタ同士の引き算

```
#define BORROW_PATTERN                                                  \
    (vec_uchar16){0x80, 0x80, 0x80,  7, 0x80, 0x80, 0x80, 0xc0,         \
                  0x80, 0x80, 0x80, 15, 0x80, 0x80, 0x80, 0xc0}

vec_llong2
sub_llong2(vec_llong2 a, vec_llong2 b)
{
    vec_int4 c;
    // spu_genbで扱えるようにvec_int4型にキャストする
    c = spu_genb((vec_int4)a, (vec_int4)b);
    // word1とword3の結果をword0とword2に移し，word1とword3のLSBを1にする
    c = spu_shuffle(c, c, BORROW_PATTERN);
    return (vec_llong2)spu_subx((vec_int4)a, (vec_int4)b, c);
}
```

また，spu_subx を行った際に桁借りが発生するかどうかについては，spu_genbx を利用することで知ることができます．spu_genbx は，2 つのベクタの要素間で引き算を行った結果から別のベクタの要素の LSB を反転した値を引いた場合に桁借りが発生するかどうかについての情報をベクタとして返します．上記のフォーマットの場合，a の各要素から b の各要素を引き，さらに c の対応する要素の LSB を反転した値を引いた際に，桁借りが発生するならば d の対応する要素に 0 を格納し，そうでないならば 1 を格納します．データの配置を工夫しておけば，これを用いることで 96bit 幅や 128bit 幅といったビット幅の大きな整数間の引き算を実現できるでしょう．

5.5.3　乗算

以下では乗算に関わる SIMD 演算を行うための型総称組み込み関数を紹介します．一口に乗算といっても，SPU の整数同士の乗算命令には，どことどこを掛け合わせるのかといったバリエーションが非常に多いため，多数の型総称組み込み関数が用意されています．また，SPU では，乗算は 16bit 同士の演算までしかハードウェアでサポートされていません．32bit 同士を掛け算したい場合は，人間が九九を駆使して筆算で 2 桁以上の掛け算を手で計算するのと同じ要領で，16bit 同士の掛け算を駆使して行うことになります．なお，蛇足ではありますが，10 進数の 2 桁同士の掛け算の結果が最大で 4 桁になるのと同じように，16bit 幅の整数同士の掛け算の結果をビット落ちなしに保持するには 32bit 幅の領域が必要になります．

spu_mul：ベクタの要素同士の掛け算

```
d = spu_mul(a, b);
```

spu_mul は，ベクタの要素同士の掛け算を行います．上記のフォーマットの場合，a の各要素と b の各要素を掛け算した結果を d の各要素に格納します．引数と戻り値の型が同じであるため，非常にシンプルでわかりやすいのですが，a および b には vec_float4 型もしくは vec_double2 型のベクタしか取ることができません．

spu_madd・spu_msub：ベクタの要素同士の掛け算 → 足し算・引き算

```
d = spu_madd(a, b, c);
d = spu_msub(a, b, c);
```

spu_madd は，ベクタの要素同士の掛け算を行った結果にさらに別のベクタの要素を足すことを行います．上記のフォーマットの場合，a の各要素と b の各要素を掛け算した結果の各要素に対して c の各要素を足した結果が d の各要素に格納されます．

一方，spu_msub は，ベクタの要素同士の掛け算を行った結果から別のベクタの要素を引くことを行います．上記のフォーマットの場合，a の各要素と b の各要素を掛け算した結果の各要素から c の各要素を引いた結果が d の各要素に格納されます．

spu_madd と spu_msub の a および b は vec_float4 型および vec_double2 型のベクタに対応しますが，spu_madd に限っては vec_short8 型のベクタにも対応します．vec_short8 型のベクタを引数に取った場合，spu_madd は，引数の各ベクタの奇数番目の要素同士の乗算を行い，

その結果に vec_int4 型のベクタを足すことを行います[11]．これは後ほど紹介する spu_mhhadd と似た挙動となります．

spu_nmadd：ベクタの要素同士の掛け算 → 足し算 → 符号反転

```
d = spu_nmadd(a, b, c);
```

spu_nmadd は，spu_madd と同様の処理を行った上で，最後に符号を反転したものを返します．vec_double2 型のベクタのみ引数に取ることができます．

spu_nmsub：ベクタの要素同士の掛け算 → 引き算 → 符号反転

```
d = spu_nmsub(a, b, c);
```

spu_nmsub は，spu_msub と同様の処理を行った上で，最後に符号を反転したものを返します．vec_float4 型と vec_double2 型のベクタのみ引数に取ることができます．

spu_mule：ベクタの要素同士の掛け算 (偶数番目同士)

```
d = spu_mule(a, b);
```

spu_mule は，16bit 幅の整数要素のベクタの偶数番目の要素同士を掛け算して 32bit 幅の整数要素のベクタを返します (spu_mule の “e” は “even” の頭文字)．上記のフォーマットの場合，a の偶数番目の各要素と，それに対応する b の偶数番目の各要素を掛けた結果が d の対応する要素に格納されます．たとえば d の word0 には a の halfword0 と b の halfword0 を掛けた結果が入り，d の word3 には a の halfword6 と b の halfword6 を掛けた結果が入ります．spu_mule での掛け算の流れを図 5.6 に示します．

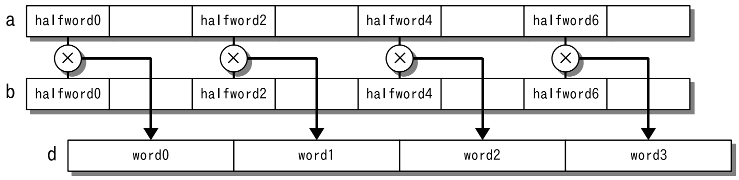

図 5.6 spu_mule での掛け算の流れ

spu_mulo：ベクタの要素同士の掛け算 (奇数番目同士)

```
d = spu_mulo(a, b);
```

spu_mulo は，16bit 幅の整数要素のベクタの奇数番目の要素同士を掛け算して 32bit 幅の整数要素のベクタを返します (spu_mulo の “o” は “odd” の頭文字)．上記のフォーマットの場合，a の奇数番目の各要素と，それに対応する b の奇数番目の各要素を掛けた結果が d の対応する要素に格納されます．たとえば d の word0 には a の halfword1 と b の halfword1 を掛けた結果

[11] 16bit 幅のデータ同士の掛け算の結果は 32bit 幅となるため．

が入り，d の word2 には a の halfword5 と b の halfword5 を掛けた結果が入ります．spu_mulo での掛け算の流れを図 5.7 に示します．

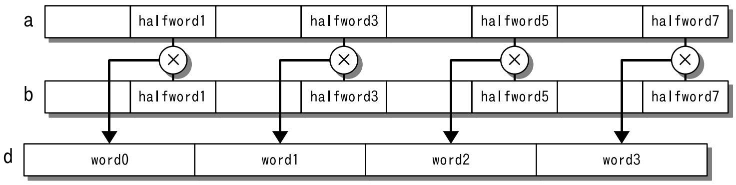

図 5.7　spu_mulo での掛け算の流れ

2 つの 16bit 幅の整数要素のベクタに対して spu_mule と spu_mulo の両方を適用することで，両方のベクタの 8 個の対応する要素同士を掛け算した結果を得ることができます．

なお，spu_mulo は，spu_mule と異なり，b に 16bit 幅のスカラの変数もしくは 10bit 幅のリテラルを取ることもできます．その場合，a の奇数番目の各要素と b の値を掛けた結果が d の対応する要素に格納されます．

spu_mulh：ベクタの要素同士の掛け算 (奇数番目 × 偶数番目) → 左シフト

```
d = spu_mulh(a, b);
```

spu_mulh は，spu_mule や spu_mulo と同様に 16bit 幅の整数要素のベクタ同士の掛け算を行いますが，引数には vec_short8 型のベクタしか取ることができません．上記のフォーマットの場合，a の偶数番目の要素と，その 1 つ右隣となるスロットに位置する b の奇数番目の要素を掛け合わせ，結果の 32bit を 16bit 左シフトして d の対応する要素に格納します．たとえば d の word0 には a の halfword0 と b の halfword1 を掛けた結果を 16bit 左シフトした結果が格納され，d の word3 には a の halfword6 と b の halfword7 を掛けた結果を 16bit 左シフトした結果が格納されます．spu_mulh での掛け算の流れを図 5.8 に示します．

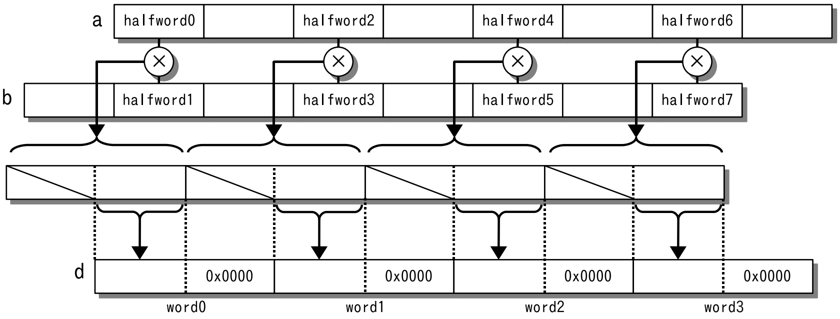

図 5.8　spu_mulh での掛け算の流れ

九九の筆算のように，引数の順番を入れ替えて spu_mulh を 2 回呼び，さらに spu_mulo を呼んだ上で，それらの結果を足し合わせると，vec_int4 型のベクタの要素同士の掛け算に相当する処理が実現します (得られる結果は下位 32bit に限定)．リスト 5.10 に vec_int4 型のベクタ a とベクタ b の各要素同士を掛け算する例を示します．

リスト 5.10 spu_mulh と spu_mulo を利用した vec_int4 型のベクタ同士の掛け算

```
vec_int4
mul32(vec_int4 a, vec_int4 b)
{
    vec_int4 tmp1, tmp2, tmp3;
    // aの各要素の上位16bitとbの各要素の下位16bitを掛け算
    tmp1 = spu_mulh((vec_short8)a, (vec_short8)b);
    // bの各要素の上位16bitとaの各要素の下位16bitを掛け算
    tmp2 = spu_mulh((vec_short8)b, (vec_short8)a);
    // aとbの各要素の下位16bit同士を掛け算
    tmp3 = (vec_int4)spu_mulo((vec_ushort8)b, (vec_ushort8)a);
    // 3つの掛け算の結果を足し合わせる
    return spu_add(tmp3, spu_add(tmp1, tmp2));
}
```

spu_mulsr：ベクタの要素同士の掛け算 (奇数番目同士) → 算術右シフト

```
d = spu_mulsr(a, b);
```

spu_mulsr は，spu_mulo と同様の演算を行った後，結果の 32bit 幅の各要素を 16bit 右に算術シフトした値を返します．spu_mulh と同様に，引数には vec_short8 型のベクタのみを取ることができます．上記のフォーマットの場合，a の奇数番目の要素と，それに対応する b の奇数番目の要素を掛けた結果を 16bit 算術右シフトした結果が d の対応する各要素に入ります．たとえば d の word0 には a の halfword1 と b の halfword1 を掛け算した結果を 16bit 算術右シフトした値が入ります．spu_mulsr での掛け算の流れを図 5.9 に示します．

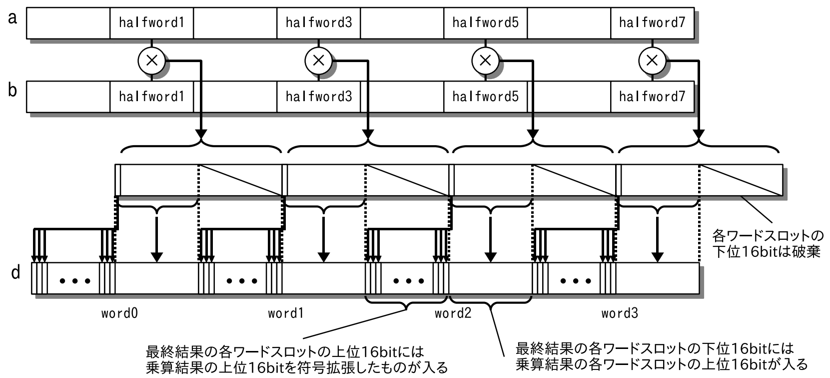

図 5.9 spu_mulsr での掛け算の流れ

spu_mhhadd：ベクタの要素同士の掛け算 (偶数番目同士) → 足し算

```
d = spu_mhhadd(a, b, c);
```

spu_mhhadd は，spu_mule と同様に vec_short8 型もしくは vec_ushort8 型のベクタの偶数番目の要素同士を掛け算し，結果に vec_int4 型もしくは vec_uint4 型のベクタを足し合わせることを行います (spu_madd の偶数要素版)．上記のフォーマットの場合，a の偶数番目の要素に，対応する b の偶数番目の要素を掛け，さらにその結果に対応する c の要素を足し，結果を d に格納します．たとえば d の word1 には a の halfword2 と b の halfword2 を掛けた値に c の

word1 を足した値が格納されます．spu_mhhadd での掛け算の流れを図 5.10 に示します．

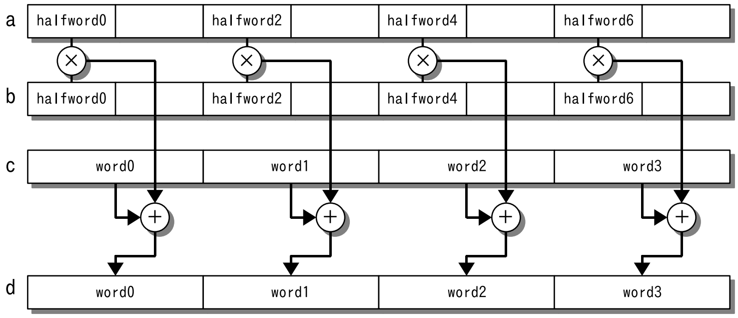

図 5.10 spu_mhhadd での掛け算の流れ

5.5.4 逆数の近似値の導出

以下では浮動小数点数の逆数の近似値を求める型総称組み込み関数を紹介します．

spu_re：ベクタの各要素の逆数

```
d = spu_re(a);
```

spu_re は，vec_float4 型のベクタの各要素の逆数の近似値を返します．この型総称組み込み関数はコンパイル時に 2 つの命令に変換されます．SPU は除算命令そのものをハードウェアで持っていないので，vec_float4 型の除算を行う場合は spu_re と spu_mul を組み合わせて実現することになります．なお，spu_re で得られる結果の精度は 12bit しかないため，精度が必要な場合はさらにニュートン・ラプソン法を使ってソフトウェアで精度を上げる必要があります．

spu_rsqrte：ベクタの各要素の逆数の絶対値の平方根

```
d = spu_rsqrte(a);
```

spu_rsqrte は，vec_float4 型のベクタの各要素の逆数の絶対値の平方根の近似値を返します．この型総称組み込み関数に関しても，spu_re と同様にコンパイル時に 2 つの命令に変換されます．また，近似値の精度も同様に 12bit であるため，そのまま使うと精度に問題がある場合はソフトウェアで精度を上げる必要があります．

なお，余談となりますが，この spu_rsqrte と spu_re はゼロ除算を発生する可能性があります．SPU ではゼロ除算が発生しても例外が発生することはなく，FPSCR と呼ばれるレジスタにフラグが立つだけで，処理はそのまま続行されます．この辺りに関する話は 5.7 で詳しく扱います．

5.5.5 バイト操作

以下ではバイト単位での操作を行う型総称組み込み関数を紹介します．いずれの型総称組み込み関数も vec_uchar16 型のベクタのみを引数として取ることができます．

spu_absd：ベクタの要素同士の差の絶対値

```
d = spu_absd(a, b);
```

spu_absd は，vec_uchar16 型のベクタの各要素間で引き算を行い，結果の絶対値を返します．上記のフォーマットの場合，a の各要素と b の各要素の差を絶対値にしたものが d の対応する各要素に格納されます．

spu_avg：ベクタの要素同士の平均

```
d = spu_avg(a, b);
```

spu_avg は，vec_uchar16 型のベクタの対応する要素同士の平均を求めます．厳密には，2 つのベクタの対応する要素同士を足して，さらにそこに 1 を加えた結果を 1bit 右シフトした結果が返ります (小数第 1 位で四捨五入を行うため)．上記のフォーマットの場合，a の各要素と b の各要素の平均値を四捨五入した結果が d の各要素に格納されます．

spu_sumb：ベクタの 4 要素単位の和

```
d = spu_sumb(a, b);
```

spu_sumb は，vec_uchar16 型の 2 つのベクタに対して，各ベクタの中で 4Byte 単位の和 (char0 から char3 までの和や char4 から char7 までの和など) を求め，求まった 8 個の 16bit 幅の結果を vec_ushort8 型で返します．上記のフォーマットの場合，d の偶数番目のスロットに b の各要素を 4 要素ずつ足し合わせたものが格納され，d の奇数番目のスロットに a の各要素を 4 要素ずつ足し合わせたものが格納されます．言葉だけの説明だと非常にわかりにくいので，spu_sumb の足し算の流れを図 5.11 に示します．結果に **b の 4Byte ごとの総和と a の 4Byte ごとの総和が交互に入る**点に注意してください．

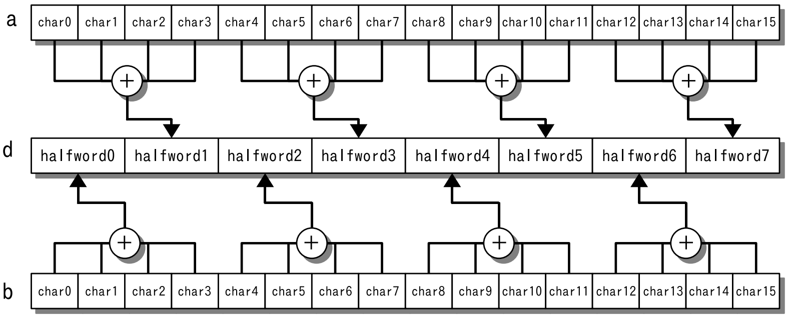

図 5.11　spu_sumb の足し算の流れ

5.5.6　比較処理

以下ではベクタ同士の比較を行う型総称組み込み関数を紹介します．SPU では比較処理の結果も汎用レジスタにビットマスクの形で格納されます．したがって，比較結果を分岐だけでなく，

さらに別の演算処理の入力として利用することができます．

spu_cmpeq：ベクタの要素同士の値が同一かどうか比較

```
d = spu_cmpeq(a, b);
```

spu_cmpeq は，2 つのベクタの各要素間の値が同一かどうかを比較して返します．比較の結果はビットマスクで返り，要素同士が等しい場合は対応する要素の全ビットが 1 になり，そうでない場合は対応する要素の全ビットが 0 になります．上記のフォーマットの場合，d の各要素は a の各要素と対応する b の各要素が等しい場合にすべてのビットが 1 となり，そうでない場合に 0 となります．a および b は vec_uchar16 や vec_int4 などの整数要素のベクタ (64bit 幅の整数要素のベクタを除く) および vec_float4 型のベクタに対応しています．また，d の型は a が 8bit 幅の要素を持つベクタだったら vec_uchar16，16bit 幅の要素を持つベクタだったら vec_ushort8，そして 32bit 幅の要素を持つベクタだったら vec_uint4 となります．

たとえば a と b が vec_uint4 型で，a の word0 と b の word0 がともに 0x12345678 だった場合，両者は一致するので d の word0 には 0xffffffff が入ります．また，a の word3 が 0x00000000 で，b の word3 が 0x00000001 だった場合，両者は一致しないので d の word0 には 0x00000000 が入ります．その際，d の型は vec_uint4 となります．

なお，a が vec_float4 型でない場合は，b に a の要素と同じ型のスカラの変数やリテラル (最大 10bit) を指定することも可能です．vec_uchar16 型のベクタや vec_char16 型のベクタとリテラルを比較する場合はリテラルの下位 8bit のみが用いられます．

spu_cmpgt：ベクタの要素同士の大小比較

```
d = spu_cmpgt(a, b);
```

spu_cmpgt は，spu_cmpeq と同じように 2 つのベクタの各要素の大小比較を行います．大小比較の結果も spu_cmpeq と同じくビットマスクで返ります．上記のフォーマットの場合，d の各要素は，a の各要素が対応する b の各要素より大きい (**“以上” ではなく “より大きい” である点に注意**) 場合にすべてのビットが 1 となり，そうでない場合に 0 となります．a および b は vec_uchar16 や vec_int4 などの整数要素のベクタ (64bit 幅の整数要素のベクタを除く) および vec_float4 型のベクタに対応しています．また，戻り値の型は，spu_cmpeq と同様に，a の型の要素のビット幅によって vec_uchar16，vec_ushort8，vec_uint4 のいずれかになります．

spu_cmpabseq：ベクタの各要素の絶対値が同一かどうか比較

```
d = spu_cmpabseq(a, b);
```

spu_cmpabseq は，vec_float4 型の 2 つのベクタの間で各要素の絶対値が同一かどうかを比較して返します．結果は vec_int4 型のベクタで返り，spu_cmpeq と同様に，一致した場合に戻り値の対応するスロットの全ビットが 1 となり，そうでない場合に 0 となります．上記のフォーマットの場合，d の各要素は，対応する a の各要素の絶対値と対応する b の各要素の絶対値が等しい場合に 0xffffffff となり，そうでない場合に 0x00000000 となります．

spu_cmpabsgt：ベクタの各要素の絶対値の大小比較

```
d = spu_cmpabsgt(a, b);
```

spu_cmpabsgt は，vec_float4 型の 2 つのベクタ間で各要素の絶対値の大小比較を行います．結果は vec_uint4 型のベクタで返り，大小比較の結果は spu_cmpgt と同様にビットマスクで返ります．上記のフォーマットの場合，d の各要素には，a の各要素の絶対値が対応する b の各要素の絶対値よりも大きい場合に 0xffffffff が格納され，そうでない場合には 0x00000000 が格納されます．

5.5.7 ビット処理

以下ではベクタのビットを数えたり，2 つのベクタをビット単位やバイト単位で合成したりする特殊な型総称組み込み関数 (ビットに対する論理演算とは別) を紹介します．

spu_cntb：ベクタの各要素の 1 が立っているビットの数の取得

```
d = spu_cntb(a);
```

spu_cntb は，vec_uchar16 型もしくは vec_char16 型のベクタについて，各要素の 1 が立っているビットの数を数え，結果を対応する要素に格納します．上記のフォーマットの場合，d の各要素には a の各要素に含まれる 1 のビットの数が入ります．たとえば a の char5 が 0x81 だった場合，d の char5 には 2 が格納され，a の char8 が 0xfe だった場合は d の char8 には 7 が格納されます．したがって，戻り値の各要素は最大でも 8 にしかなりません．大して複雑ではありませんが，リスト 5.11 の記述における spu_cntb の挙動を図 5.12 に示します．

リスト 5.11 spu_cntb の挙動を確認するコードの例

```
vec_uchar16 a, d;
a = (vec_uchar16){0x01, 0x02, 0x03, 0x05, 0x07, 0x0b, 0x0d, 0x11,
                  0x13, 0x17, 0x1d, 0x1f, 0x25, 0x29, 0x2b, 0x2f};
d = spu_cntb(a);
```

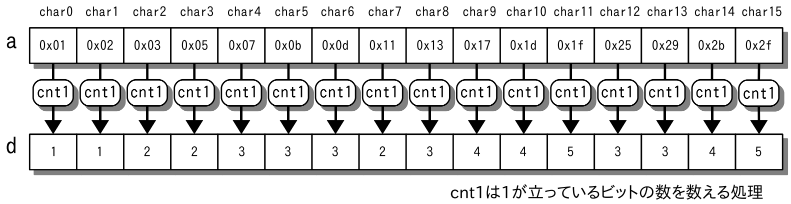

図 5.12 spu_cntb の挙動の例

spu_cntlz：ベクタの各要素の MSB からの連続した 0 の数の取得

```
d = spu_cntlz(a);
```

spu_cntlz は，32bit 幅の要素のベクタに対して，各要素をビット列とみなし，MSB 側から見て初めて 1 のビットが現れるまでに何個の 0 があったかを vec_uint4 型のベクタの対応する要素に格納して返します．上記のフォーマットの場合，a の各要素について，MSB から LSB にかけて 1 が初めて現れるまでに何個 0 のビットがあったかを数えて d の対応する各要素に格納します．たとえば a の word1 が 0x0031f020 だった場合，word1 のビット 10 が最初の 1 となるため，その左に 10 個の 0 があるということになり，d の word1 には 10 が格納されます．また，a の word3 が 0x80000000 だった場合，MSB が 1 なので，d の word3 には 0 が格納されます．

spu_gather：ベクタの各要素の LSB の抽出

```
d = spu_gather(a);
```

spu_gather は，ベクタの各要素の LSB を取り出して連接したものを vec_uint4 型のベクタの word0 の位置に右詰めして返します．上記のフォーマットの場合，a が vec_uchar16 型のベクタだったとすると，d の word0 のビット 31 (LSB) に a の char15 の LSB の値が格納され，d の word0 のビット 16 に a の char0 の LSB の値が格納されます．a は 8bit 幅，16bit 幅および 32bit 幅の要素を持つベクタに対応します．また，戻り値の word0 の余ったビットや，word1 から word3 までのビットはすべて 0 となります．

図 5.13 に a が 8bit 幅の要素のベクタの場合の spu_gather の挙動を，図 5.14 に a が 16bit 幅の要素のベクタの場合の spu_gather の挙動をそれぞれ示します．

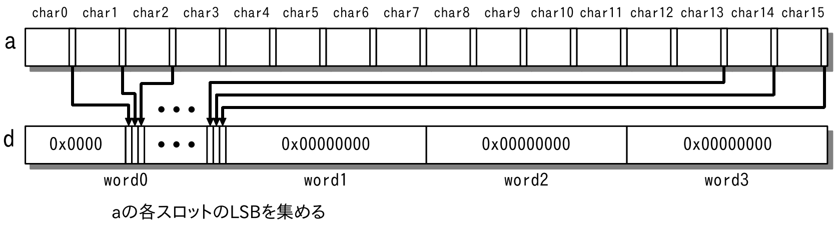

図 5.13　spu_gather で 8bit 幅の要素のベクタを引数にした場合の流れ

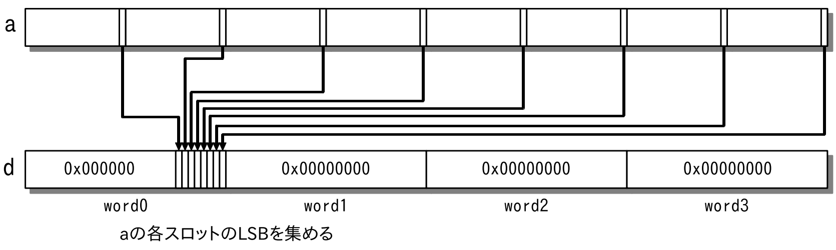

図 5.14　spu_gather で 16bit 幅の要素のベクタを引数にした場合の流れ

spu_sel：ベクタをパターンを用いてビット単位で混合

```
d = spu_sel(a, b, p);
```

spu_sel は，同じ型の 2 つのベクタの値をビット単位で混ぜて，同じ型の新たなベクタを生成します．2 つのベクタ以外に，それらをどう混ぜるかを指示するためのパターン用のベクタを必要とします．上記のフォーマットの場合，a，b および p の各ベクタを 128bit のビット列とみなして，p のビットが 0 だったら，それに対応する a のビットを取り出し，1 だったら b のビットを取り出して，d の対応するビットに格納します．a および b は SPE で利用できる 10 種類すべてのベクタの型に対応しています．また，パターン用のベクタ p は vec_uchar16，vec_ushort8，vec_uint4 および vec_ullong2 に対応しており，a および b の各要素の幅に合わせて適切なもの (a や b と要素の幅が同じ型のベクタ) を取ります．

リスト 5.12 のコードを実行した際の spu_sel の挙動を図 5.15 に示します．スペースの都合上，図では MSB 側と LSB 側の一部のみを示しており，間のビットについては省略していますが，実際には d の全 128bit に対して，p の値により a もしくは b の対応するビットの値が格納されます．

リスト 5.12　spu_sel の挙動を確認するコードの例

```
vec_uint4 a, b, d, p;
p = (vec_uint4){0x6f01382d, 0x00f197ca, 0xe124337d, 0xd230416e};
  ...
d = spu_sel(a, b, p);
```

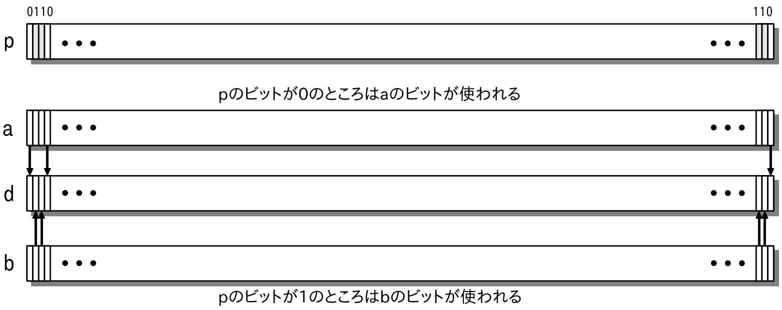

図 5.15　spu_sel の挙動の例

6.4 で詳しく紹介しますが，spu_sel は比較演算子の結果と組み合わせるなどして分岐命令を除去するのに使えたりします．また，spu_sel の振る舞いは論理式に変換すると d = (a & ~p) | (b & p) と等価となるため，3 入力の論理式とみなして論理演算の命令数を減らすといった特殊な用途にも利用できます．

spu_shuffle : ベクタをパターンを用いてバイト単位で混合

```
d = spu_shuffle(a, b, p);
```

spu_shuffle は，同じ型の 2 つのベクタからバイト単位で値を取り出して混ぜ合わせ，同じ型の新たなベクタを生成します．混ぜ合わせる 2 つのベクタは SPE で扱える 10 種類すべての型のベクタに対応し，spu_sel と同様に，どう混ぜ合わせるかを指示するためのパターン用のベクタ (vec_uchar16 型) を別途必要とします．また，パターンに特殊な値を与えると，混ぜ合わせる対象の 2 つのベクタの一部ではなく，0x00，0xff，0x80 のいずれかの定数を結果に格納することができます．上記のフォーマットの場合，パターンとなるベクタ p の n 個目の要素の値を m としたとき，d の nByte 目には表 5.3 に示す値が格納されます．

表 5.3　spu_shuffle でのパターンの値と戻り値の関係

m の値	戻り値の nByte 目に入る値
0 以上 15 以下	ベクタ a のスロット char m の値
16 以上 31 以下	ベクタ b のスロット char (m - 16) の値
128 (0x80) 以上 192 (0xc0) 未満	0x00
192 (0xc0) 以上 224 (0xe0) 未満	0xff
224 (0xe0) 以上	0x80

たとえば p の char0 が 1 だったら d の char0 には a の char1 の値が格納され，p の char3 が 20 だったら d の char3 には b の char4 の値が格納されます[*12]．また，p の char14 が 0x80 だった場合は，d の char14 には 0 が格納されます．

図 5.16 に，リスト 5.13 の記述に対応した spu_shuffle の挙動を示します．リスト 5.13 の記述では，vec_uchar16 型のベクタ a とベクタ b を，ベクタ p のパターンを用いて混ぜ合わせています．この混ぜ合わせ方自体に何か特別な意味があるわけではありませんが，ベクタ p の char8 から char11 までの 4 要素の値が 128 以上であるため，ベクタ d の char8 から char11 までのスロットには定数が入り，p のそれ以外の要素は 128 未満なので，残りのベクタ d のスロットには a もしくは b の各バイトの値が格納されます．なお，図の a の上および b の下に書いてある括弧内の数字は，p の各バイトの値 (厳密にはそれが 128 未満のときに 32 で割った余り) によって選ばれるバイトの番号になります．

リスト 5.13　spu_shuffle の挙動を確認するコードの例

```
vec_uchar16 a, b, d, p;
p = (vec_uchar16){15,   14,  13,  12, 31, 30, 29, 28,
                  128, 128, 192, 192, 28, 29, 30, 31};
a = (vec_uchar16){0xa0, 0xa1, 0xa2, 0xa3, 0xa4, 0xa5, 0xa6, 0xa7,
                  0xa8, 0xa9, 0xaa, 0xab, 0xac, 0xad, 0xae, 0xaf};
b = (vec_uchar16){0xb0, 0xb1, 0xb2, 0xb3, 0xb4, 0xb5, 0xb6, 0xb7,
                  0xb8, 0xb9, 0xba, 0xbb, 0xbc, 0xbd, 0xbe, 0xbf};
d = spu_shuffle(a, b, p);
```

[*12] 厳密にいうと，a や b の値が結果に格納されるのは m の値が 128 未満で，m の下位 5bit が表 5.3 に示した条件に合致する場合になります．

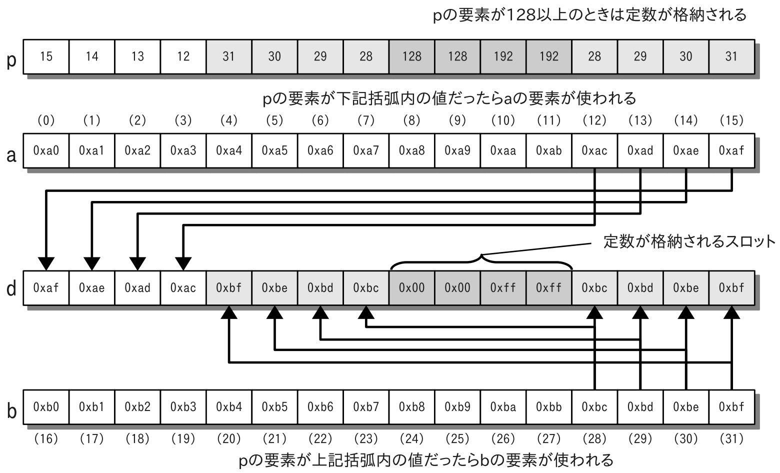

図 5.16 spu_shuffle の挙動の例

5.5.8 ビットマスク生成

以下ではビットマスクの生成を行う型総称組み込み関数を紹介します.

spu_maskb : スカラからビットマスク生成 (16bit → 128bit)

```
d = spu_maskb(a);
```

spu_maskb は,スカラ (16bit か 32bit の変数もしくは 16bit のリテラル) の下位 16bit について,各ビットをビット単位で 8 個ずつ並べて連接し,vec_uchar16 型のビットマスクを生成します.上記のフォーマットの場合,d の各要素に a の下位 16bit の各ビットの値を 8 個並べたものが格納されます.たとえば a の下位 16bit について,ビット 0 が 1 だったら d の char0 に 0xff が格納され,a のビット 7 が 0 だったら d の char7 に 0x00 が格納されます.

図 5.17 に spu_maskb のビットマスクの生成の様子を模式化したものを示します.スペースの都合上,d の char0 および char15 の値の部分のみ図示してありますが,実際には a の下位 16bit すべてについて同様の処理が行われます.

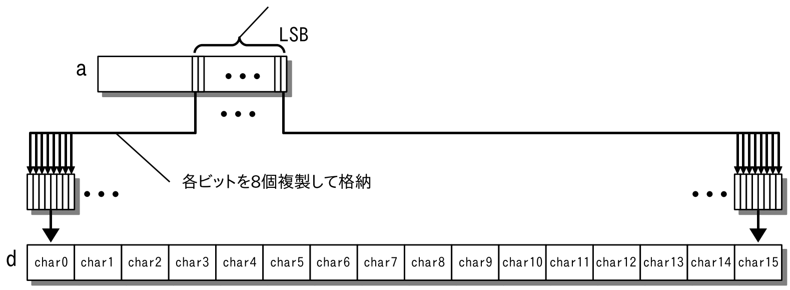

図 5.17 spu_maskb によるビットマスクの生成

spu_maskh：スカラからビットマスク生成 (8bit → 128bit)

```
d = spu_maskh(a);
```

spu_maskh は，8bit，16bit，32bit のいずれかのスカラの変数の下位 8bit について，各ビットをビット単位で 16 個ずつ並べて連接し，vec_short8 型のビットマスクを生成します．上記のフォーマットの場合，d の各要素に a の各ビットを 16 個並べたものが格納されます．たとえば a の下位 8bit の各ビットについて，ビット 0 が 1 だったら d の halfword0 に 0xffff が格納され，ビット 4 が 0 だったら d の halfword4 に 0x0000 が格納されます．

図 5.18 に spu_maskh のビットマスクの生成の様子を模式化したものを示します．スペースの都合上，d の halfword0 および halfword7 の値の部分のみ図示してありますが，実際には a の下位 8bit すべてについて同様の処理が行われます．

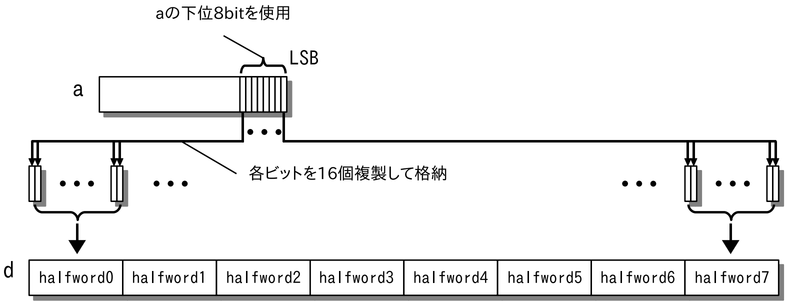

図 5.18　spu_maskh によるビットマスクの生成

spu_maskw：スカラからビットマスク生成 (4bit → 128bit)

```
d = spu_maskw(a);
```

spu_maskw は，8bit，16bit，32bit のいずれかのスカラの変数の下位 4bit について，各ビットをビット単位で 32 個ずつ並べて連接し，vec_uint4 型のビットマスクを生成します．上記のフォーマットの場合，d の各要素に a の下位 4bit の各ビットの値を 32 個並べたものが格納されます．たとえば a の下位 4bit について，ビット 0 が 1 だったら d の word0 に 0xffffffff が格納され，ビット 3 が 0 だったら d の word3 に 0x00000000 が格納されます．

図 5.19 に spu_maskw のビットマスクの生成の様子を模式化したものを示します．

5.5.9　論理演算

以下ではベクタ間 (もしくはベクタ内) でのビット単位での論理演算を行う型総称組み込み関数を紹介します．論理演算系の型総称組み込み関数は，spu_orx を除いたすべての関数で全種類のベクタを引数に取ることができます．戻り値は引数のベクタと同じ型となります．

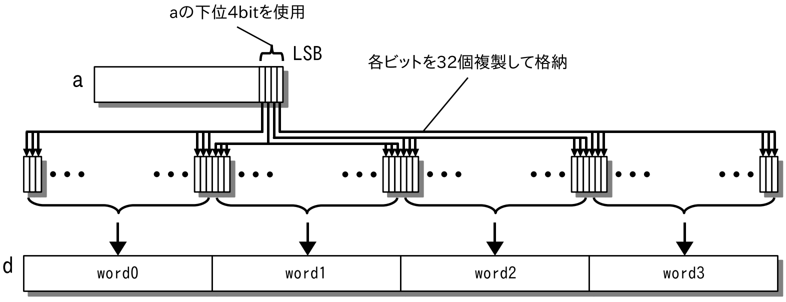

図 5.19 spu_maskw によるビットマスクの生成

spu_and・spu_nand：ベクタ同士のビット単位での AND・NAND

```
d = spu_and(a, b);
d = spu_nand(a, b);
```

spu_and は，2 つのベクタの各ビット間の AND を返します．また，spu_nand は，2 つのベクタの各ビット間の NAND (AND の結果の反転) を返します．spu_and は，a が 8bit 幅から 32bit 幅の整数要素のベクタのときに限り，b にスカラの変数もしくはリテラルを取ることができます．その場合，d には a の各要素に b の値を AND した結果が格納されます．

spu_or・spu_nor：ベクタ同士のビット単位での OR・NOR

```
d = spu_or(a, b);
d = spu_nor(a, b);
```

spu_or は，2 つのベクタの各ビット間の OR を返します．また，spu_nor は，2 つのベクタの各ビット間の NOR (OR の結果の反転) を返します．spu_and と同じく，spu_or は，a が 8bit 幅から 32bit 幅の整数要素のベクタのときに限り，b にスカラの変数もしくはリテラルを取ることができます．その場合，d には a の各要素に b の値を OR した結果が格納されます．

spu_xor・spu_eqv：ベクタ同士のビット単位での XOR・XNOR

```
d = spu_xor(a, b);
d = spu_eqv(a, b);
```

spu_xor は，2 つのベクタの各ビット間の XOR を返します．また，spu_eqv は，2 つのベクタの各ビット間を比較し，同一の場合は結果の対応するビットに 1 を返し，そうでない場合は 0 を返します．したがって，spu_eqv は，ベクタの各ビット間の XNOR を返すものと考えることができます．spu_and と同じく，spu_xor は，a が 8bit 幅から 32bit 幅の整数要素のベクタのときに，b にスカラの変数もしくはリテラルを指定することができます．その場合，d には a の各要素に b の値を XOR した結果が格納されます．

spu_andc：ベクタの一方をビット反転 → ビット単位で AND

```
d = spu_andc(a, b);
```

spu_andc は，2 つのベクタについて，一方のベクタのビットを反転した結果と，もう一方のベクタとのビット単位の AND を返します．上記のフォーマットの場合，b のビットを反転させてから a と AND を取り，その結果を d に格納します．反転するのが b の方である点に注意してください．論理式にすると d = a & ~b となります．

spu_orc：ベクタの一方をビット反転 → ビット単位で OR

```
d = spu_orc(a, b);
```

spu_orc は，2 つのベクタについて，一方のベクタのビットを反転した結果と，もう一方のベクタとのビット単位の OR を返します．上記のフォーマットの場合，b のビットを反転させてから a と OR を取り，その結果を d に格納します．spu_andc と同様に，反転するのが b の方である点に注意してください．論理式にすると d = a | ~b となります．

spu_orx：ベクタの 4 つのワードスロットの OR

```
d = spu_orx(a);
```

spu_orx は，vec_int4 型もしくは vec_uint4 型のベクタを引数に取り，4 つのワードスロット (word0 から word3 まで) の OR を取って，結果を word0 に格納する特殊なビット演算です．結果の word1 から word3 までには 0 が格納されます．上記のフォーマットの場合，d の word0 に a の word0，word1，word2 および word3 の 4 つの 32bit 幅の値の OR を取ったものが格納されます．図 5.20 に spu_orx の処理の流れを示します．

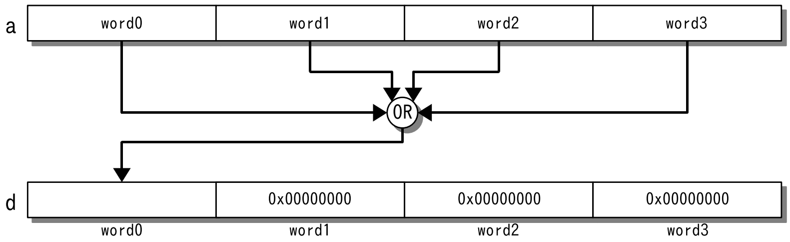

図 5.20 spu_orx の処理の流れ

5.5.10 シフト・ローテート

以下ではベクタのシフトもしくはローテートを行う型総称組み込み関数を紹介します．シフト・ローテートの型総称組み込み関数は，名前が非常に長く，名前から機能が直感的にわかりにくい部分があるので，個々を説明する前に，まずどのような操作ができるかを整理しておきます．

SPE のシフト・ローテート処理には左ローテート (rl)，左シフト (sl)，論理右シフト (rlmask)，算術右シフト (rlmaska) の 4 種類が用意されています．そのうち算術右シフトを除いた 3 種類の

処理については，以下の 3 つのパターンが存在します (算術右シフトについては 1 つ目のパターンのみ).

- ベクタの各要素内でビット単位でシフト・ローテートするもの

 対象となるベクタの各要素を，別のベクタの対応する各要素の値の分だけシフトもしくはローテートします．例として，図 5.21 に 32bit 幅の要素のベクタを要素内で左シフトした際の動きを示します．図では，ベクタ a の各要素をベクタ s の各要素の値の分だけ要素内で左シフトして，結果をベクタ d の対応する要素に格納しています．たとえば d の word0 には a の word0 の値を s の word0 に格納されている値だけ左シフトした値が格納されます.

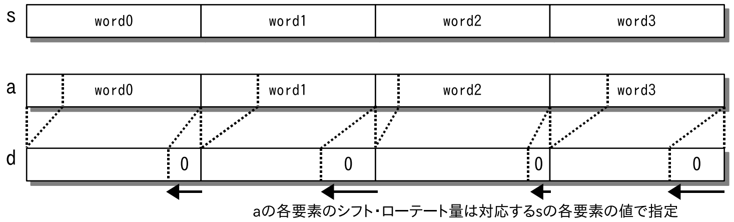

図 5.21　ベクタを要素内で左シフトした際の動き

- ベクタ全体をビット単位でシフト・ローテートするもの

 対象となるベクタを 128bit 幅のビット列とみなして，別途スカラの変数やリテラルで指定した値の分のシフトもしくはローテートを行います．最大 7bit まで 1bit 刻みでシフト・ローテートさせることが可能です．例として，図 5.22 にベクタ全体を 4bit 左ローテートさせた際の動きを示します.

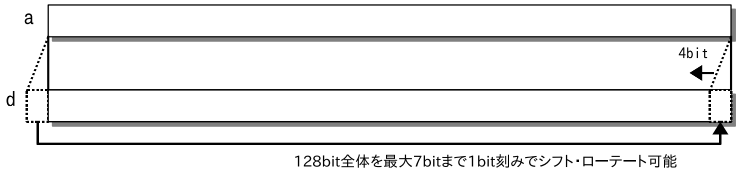

図 5.22　ベクタ全体を 4bit 左ローテートさせた際の動き

 このパターンに対応する動きをする型総称組み込み関数は，関数名の末尾に “qw” という文字列がつきます.

- ベクタ全体をバイト単位でシフト・ローテートするもの

 対象となるベクタを 16Byte のバイト列とみなして，別途スカラの変数やリテラルで指定したバイト数分のシフトもしくはローテートを行います．例として，図 5.23 にベクタを 2Byte 右シフトさせた際の動きを示します.

 このパターンに対応する動きをする型総称組み込み関数は，関数名の末尾に “qwbyte” もしくは “qwbytebc” という文字列がつきます．`qwbyte` のつくものと `qwbytebc` のつくものとでは，バイト数の指定の仕方が異なります．`qwbyte` の方はバイト数そのものを指定し，

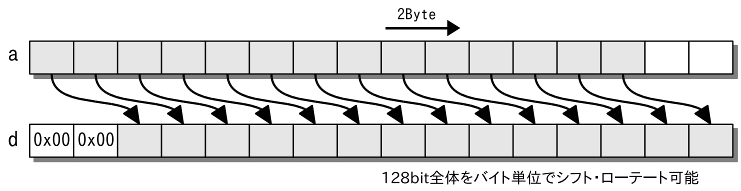

図 5.23　ベクタ全体を 2Byte 右シフトさせた際の動き

qwbytebc の方はビット数を使ってバイト数を指定します[*13].

ベクタ全体のシフト・ローテートに関しては，ビット単位のシフト・ローテートとバイト単位のシフト・ローテートを組み合わせることで，任意の大きさのシフト・ローテートを 1bit 刻みで行うことが可能となります.

以下では左ローテート，左シフト，論理右シフトの順で紹介していきます.

spu_rl：ベクタの各要素を左ローテート

```
d = spu_rl(a, c);
```

spu_rl は，ベクタの要素ごとの左ローテートを行います．ローテートできるのは 16bit 幅の要素のベクタと 32bit 幅の整数要素のベクタに限られます．各要素のローテート値は，スカラの変数やリテラルで一括指定可能な他，ベクタを用いて要素ごとに個別に指定することも可能です．ローテート対象が 16bit 幅の要素のベクタのときは，指定した数の下位 4bit がローテート値として用いられ，32bit 幅の要素のベクタのときは，指定した数の下位 5bit がローテート値として用いられます．上記のフォーマットの場合，d の各要素には，a の各要素を c の対応する各要素の値の分だけ左ローテートした結果が格納されます.

spu_rlqw：ベクタ全体をビット単位で左ローテート

```
d = spu_rlqw(a, c);
```

spu_rlqw は，任意の型のベクタを 128bit 幅のビット列とみなしてベクタ全体をビット単位で左ローテートします．ローテートする大きさは int 型の変数かリテラルで指定します．その際，ローテートの大きさは下位 3bit のみが参照され，最大 7bit までの左ローテートが可能です．それ以上左ローテートさせたい場合は spu_rlqwbyte や spu_rlqwbytebc を使います．上記のフォーマットの場合，a を型に関係なく 128bit 幅のひと続きの変数とみなして，それを c で示されるビット数分左ローテートした結果が d に格納されます.

spu_rlqwbyte・spu_rlqwbytebc：ベクタ全体をバイト単位で左ローテート

```
d = spu_rlqwbyte(a, c);
d = spu_rlqwbytebc(a, c);
```

spu_rlqwbyte と spu_rlqwbytebc は，ともに任意の型のベクタ全体をバイト単位で左

*13 指定した値を 8 で割った商をバイト数とします．ただし，詳細は後述しますが，一部ややこしい例外があります.

ローテートします．ローテートする大きさは，spu_rlqw と同様に，int 型の変数かリテラル (spu_rlqwbytebc は変数のみ) で指定します．上記のフォーマットの場合，いずれもベクタ a をスカラ c から求まるバイト数分左ローテートした結果が d に格納されます．ただし，両者の間では c の扱いに違いがあります．spu_rlqwbyte では，c の下位 4bit をローテートするバイト数として用います．一方，spu_rlqwbytebc では，c の下位 7bit の上位 4bit をローテートするバイト数として用います．したがって，spu_rlqwbytebc では，c の下位 7bit を 8 で割った商に相当するバイト数分左ローテートすることになります．spu_rlqwbytebc の存在価値に疑問を抱かれるかもしれませんが，リスト 5.14 のように，ベクタ a を cbit (c は 0 から 127 までの任意の値を取るスカラ変数) 左ローテートさせる処理をシンプルに記述するのに使えます．

リスト 5.14 spu_rlqw と spu_rlqwbytebc を組み合わせた任意のビット単位の左ローテート

```
d = spu_rlqwbytebc(spu_rlqw(a, c), c);
```

spu_sl：ベクタの各要素を左シフト

```
d = spu_sl(a, c);
```

spu_sl は，ベクタの要素ごとの左シフトを行います．spu_rl と同様に，各要素のシフト量はスカラの変数やリテラルによる一括指定およびベクタを用いた要素ごとの個別指定が可能です．spu_rl と異なり，左シフト対象が 16bit 幅の要素のベクタのときは指定した数の下位 5bit が左シフト量として用いられ，32bit 幅の要素のベクタのときは，指定した数の下位 6bit が左シフト量として用いられます．spu_rl よりも 1bit 多くなっているのは，完全なシフトアウトを実現するためです．上記のフォーマットの場合，d の各要素には a の各要素を c の対応する各要素の値の分だけ左シフトした結果が格納されます．

spu_slqw：ベクタ全体をビット単位で左シフト

```
d = spu_slqw(a, c);
```

spu_slqw は，spu_rlqw と同じように任意の型のベクタを 128bit 幅のビット列とみなしてベクタ全体を左シフトします．spu_rlqw と同様に，左シフト量は int 型の変数かリテラルで指定します．その際，int 型で指定する左シフト量は，下位 3bit のみが参照され，最大 7bit までの左シフトが可能となっています．上記のフォーマットの場合，a を型に関係なく 128bit 幅のひと続きの変数とみなして，それを c で示されるビット数分左シフトした結果が d に格納されます．

spu_slqwbyte・spu_slqwbytebc：ベクタ全体をバイト単位で左シフト

```
d = spu_slqwbyte(a, c);
d = spu_slqwbytebc(a, c);
```

spu_slqwbyte と spu_slqwbytebc は，ベクタをバイト単位で左シフトします．spu_slqw と同様に，シフト量は int 型の変数かリテラル (spu_slqwbytebc は変数のみ) で指定します．上記のフォーマットの場合，いずれもベクタ a をスカラ c から求まるバイト数分左シフトした

結果が d に格納されます．ただし，spu_rlqwbyte と spu_rlqwbytebc の関係と同じように，両者の間では c の扱いに違いがあります．spu_slqwbyte では，シフトするバイト数は c の下位 5bit を用います (完全にシフトアウトするよう 16Byte の左シフトを実現するため)．一方，spu_slqwbytebc では，c の下位 8bit の上位 5bit をシフトするバイト数として用います．これにより，spu_slqwbytebc をリスト 5.15 のように用いることで，spu_rlqwbytebc と同様に，ベクタ a を cbit (c は 0 から 128 までの任意の値を取るスカラ変数) 左シフトさせる処理をシンプルに記述することができます．

リスト 5.15　spu_slqw と spu_slqwbytebc を組み合わせた任意のビット単位の左シフト

```
    d = spu_slqwbytebc(spu_slqw(a, c), c);
```

spu_rlmask：ベクタの各要素を論理右シフト

```
d = spu_rlmask(a, c);
```

spu_rlmask は，名前からまったく想像がつきませんが，ベクタの各要素をビット単位で論理右シフトします．右シフトできるのは 16bit 幅の要素のベクタと 32bit 幅の整数要素のベクタに限られます．spu_sl などと同じく，各要素がどれだけ右シフトするかは，スカラの変数やリテラルによる一括指定およびベクタを用いた要素ごとの個別指定が可能です．ただし，これがややこしいのですが，**シフトの大きさは負の数にして指定しなければなりません**．上記のフォーマットの場合，ベクタ a の各要素を，c の各要素 (もしくはスカラの値) の符号を反転させた値の下位数ビット (右シフト対象が 16bit 幅の要素のベクタのときは下位 5bit，32bit 幅の要素のベクタのときは下位 6bit) 分右シフトした結果がベクタ d の各要素に入ります．たとえば a が vec_uint4 型のベクタで c の word0 が −3 だった場合，d の word0 には a の word0 が 3bit 論理右シフトされた値が格納されます．

spu_rlmaskqw：ベクタ全体をビット単位で論理右シフト

```
d = spu_rlmaskqw(a, c);
```

spu_rlmaskqw は，spu_slqw と同じように任意の型のベクタを 128bit 幅のビット列とみなしてベクタ全体を論理右シフトします．spu_slqw と同じように，シフトする大きさは int 型の変数かリテラルで指定します．最大で 7bit までの論理右シフトが可能ですが，シフト量としては spu_rlmask と同じように符号を反転させた値を設定する必要があります．上記のフォーマットの場合，a をベクタの型に限らず 128bit 幅のひと続きの変数とみなして，それを c の値の符号を反転させた値の下位 3bit 分論理右シフトした結果が d に格納されます．たとえば c が −4 の場合，ベクタ a 全体を 4bit 右に論理シフトした結果が d に格納されます．

spu_rlmaskqwbyte・spu_rlmaskqwbytebc： ベクタ全体をバイト単位で論理右シフト

```
d = spu_rlmaskqwbyte(a, c);
d = spu_rlmaskqwbytebc(a, c);
```

spu_rlmaskqwbyte と spu_rlmaskqwbytebc は，spu_slqwbyte などと同じようにベクタをバイト単位で論理右シフトします．シフトする大きさも，spu_slqwbyte と同様に，int 型の変数かリテラル (spu_rlmaskqwbytebc は変数のみ) で指定します．上記のフォーマットの場合，いずれもベクタ a をスカラ c から求まるバイト数分右シフトした結果が d に格納されます．ただし，両者の間では c の扱いに違いがあります．

spu_rlmaskqwbyte では，c の符号を反転させた値の下位 5bit を論理右シフトするバイト数として扱います．たとえば c に −3 を指定した場合，符号を反転して下位 5bit を取り出すことになるので，3Byte 分論理右シフトされます．

一方，spu_rlmaskqwbytebc での論理右シフト量の計算はかなり複雑で，spu_rlqwbytebc や spu_slqwbytebc のように直感的に挙動を理解するのは困難です．spu_rlmaskqwbytebc においても負の数を用いて論理右シフト量を指定する点は他の spu_rlmask 系の型総称組み込み関数と同じなのですが，負の値の使い方にクセがあります．上記のフォーマットの場合，c の下位 8bit の上位 5bit 分を切り出して，その符号を反転させた数だけ a を論理右シフトした結果が d に格納されます．たとえば c に −10 を与えた場合，−10 の下位 8bit は 0xf6 となり，その上位 5bit は 2 進数で 11110 となります．これの符号を反転させる (2 の補数を求める) と 2 となります．よって，ここでは a を 2Byte 右シフトした結果が d に格納されることになります．別の言い方をすると，c を 3bit 右に算術シフトして符号を反転した値に相当するバイト数分論理右シフトすることになります．

どちらで考えてもよいのですが，いずれにせよ，−9 から −16 までのいずれかの値を c に設定すると 2Byte 論理右シフトし，−17 から −24 までのいずれかの値を c に設定すると 3Byte 論理右シフトするといった挙動は，“8 で割って符号を反転した結果に相当するバイト数分論理右シフトする” とは扱えないのが辛いところです．このような理由から，spu_rlmaskqwbytebc では，ベクタ全体のビット単位の論理右シフトとの組み合わせによる任意のビット数の論理右シフトを，spu_rlqwbytebc や spu_slqwbytebc ほどシンプルに記述することができません．文献 [2] にも掲載されていますが，spu_rlmaskqwbytebc で 0 から 128 の任意のビット数に対応した論理右シフトを記述した場合，リスト 5.16 のようになります (c はスカラ変数)．このように記述すると，c で指定した値の符号を反転したビット数分 a を論理右シフトした結果が d に格納されるようになります．

リスト 5.16 spu_rlmaskqw と spu_rlmaskqwbytebc を組み合わせた任意のビット単位の論理右シフト

```
    d = spu_rlmaskqwbytebc(spu_rlmaskqw(a, c), 7 + c);
```

spu_rlmaska：ベクタの各要素を算術右シフト

```
d = spu_rlmaska(a, c);
```

spu_rlmaska は，spu_rlmask とほとんど同じですが，シフトの仕方が論理右シフトではなく算術右シフトである点が異なります．

spu_rlmask で負の値を用いる理由

なぜ spu_rlmask では右シフトする量を負の値を用いて指定するのでしょうか．組み込み関数の名前や文献 [3] の解説などから察するに，どうも SPU は内部的に論理右シフトを実装しておらず，代わりに左ローテートさせた上でローテートしなかったビットをマスクするという処理を行っているように思えます．そう考えると，負の値を用いて右シフト量を指定するという状況も合点がいきます．

たとえば，-3 を signed char 型として 16 進数に変換すると 0xfd となります．このとき，下位 4bit を取り出すと 13 となり，16bit 幅の要素のベクタの場合，13bit 左ローテートすることになるので，3bit 右ローテートしたのと同じ結果が得られます．ローテートされずに残った上位 3bit を 0 で埋めれば，結果的に右シフトとなります．同様に -3 の 16 進数表現の下位 5bit を取り出すと 29 となり，32bit 幅の要素のベクタを 29bit 左ローテートすることで 3bit 右ローテートしたのと同じ結果が得られ，同様に上位をマスクすることで 3bit の右シフトと同じ結果を得ることができます．

このように，spu_rlmask については何となく理解はできるのですが，一方で spu_rlmaskqw の方は上記のような計算ではローテートの量が合わず，なぜ負の数を用いて指定する必要があるのかよくわかりませんでした．また，spu_rlmaskqwbytebc に至ってはややこしすぎて考える気も起きません．

何にせよ，ハードウェア的な事情でこのような呼び出し方となっているのであれば仕方ないとは思うのですが，この異常に長い組み込み関数名はもう少しどうにかならなかったのだろうかとはプログラムを書くたびに思います…．

5.5.11 キャスト

SPU ではキャストも SIMD 演算で複数の数に対してまとめて行うことが可能になっています．

spu_convtf：ベクタの各要素を整数から単精度の浮動小数点数にキャスト

```
d = spu_convtf(a, s);
```

spu_convtf は，vec_int4 型もしくは vec_uint4 型の各要素の値を float 型に変換した上で，2 を指定回数べき乗した値で割った結果を vec_float4 型のベクタの各要素に格納して返します．普通に C 言語で int から float へのキャスト処理を記述した場合，コンパイラによって，この型総称組み込み関数に対応する命令が挿入されます．べき乗回数には 0 から 127 までの値が指定可能です．上記のフォーマットの場合，ベクタ a の各要素を float 型に変換し，それを $2^{\mathtt{s}}$ で割った結果が d の対応する各要素に格納されます．s を 0 にすると，通常の C 言語のキャスト処理と同じことが行えます．整数型のベクタを固定小数点数として用いている場合は，小数部の桁数を s に指定することで，うまく浮動小数点数に変換できそうです．

spu_convts：ベクタの各要素を単精度の浮動小数点数から符号つきの整数にキャスト

```
d = spu_convts(a, s);
```

spu_convts は，spu_convtf とは逆に，vec_float4 型のベクタの各要素に 2 を指定回数べき乗した値を掛けた上で int 型に変換した結果を vec_int4 型のベクタの各要素に格納して返します．こちらについても，float 型から int 型へのキャストを C 言語で記述した際に，コンパイラによってこの型総称組み込み関数に対応する命令が挿入されます．べき乗回数には 0 から 127 までの値が指定可能です．各 int 型の変換結果が $2^{31}-1$ より大きくなってしまった場合は $2^{31}-1$ に，$-(2^{31})$ より小さくなってしまった場合は $-(2^{31})$ にそれぞれ強制的に設定されます．上記のフォーマットの場合，ベクタ a の各要素に $2^{\mathtt{s}}$ を掛け，それを int 型に変換した結果が d の対応する各要素に格納されます．s を 0 にすると，通常の C 言語のキャストと同じことが行えます．vec_float4 の各要素を固定小数点数に変換したい場合，固定小数点数の小数部の桁数を s に指定することで，ちょうどよい数が得られそうです．

spu_convtu：ベクタの各要素を単精度の浮動小数点数から符号なしの整数にキャスト

```
d = spu_convtu(a, s);
```

spu_convtu は，spu_convts と同様に，vec_float4 型のベクタの各要素に 2 を指定回数べき乗した値を掛けた上で unsigned 型に変換した結果を vec_uint4 型のベクタの各要素に格納して返します．変換結果が vec_int4 ではなく vec_uint4 となるため，spu_convts とは丸めの仕方が異なり，変換結果が $2^{32}-1$ より大きくなってしまったら $2^{32}-1$ に，負の数だったら 0 にそれぞれ強制的に設定されます．

spu_extend：ベクタの一部要素のビット幅を倍に拡張

```
d = spu_extend(a);
```

spu_extend は，vec_char16 型，vec_short8 型，vec_int4 型および vec_float4 型のベクタを引数に取り，それぞれ vec_short8 型，vec_int4 型，vec_llong2 型および vec_double2 型に変換して返します．その際，引数が整数要素のベクタの場合，奇数番目の要素を符号拡張して倍の幅にした値が，結果の対応する要素に格納されます．この型総称組み込み関数に対応する命令は，C 言語のソースコード上で short 型の変数を int 型にキャストする部分などでコンパイラによって挿入されます．

一方，引数が vec_float4 型の場合は，他の型のときと異なり，偶数番目の要素が double 型のフォーマットに変換されて，結果の対応する要素に格納されます．

spu_roundtf：ベクタの要素を倍精度から単精度に丸める

```
d = spu_roundtf(a);
```

spu_roundtf は，spu_extend とは逆に，vec_double2 型のベクタを vec_float4 型のベク

タに変換する処理を行います．vec_double2 で doubleword0 および doubleword1 に入っていた値は，それぞれ結果の word0 および word2 に入り，結果の word1 と word3 には 0 が入ります．

長々と演算系の型総称組み込み関数の紹介をしましたが，5.5.11 までで演算と直結しているものは出尽くしたと思います[*14]．一度に全部覚えるなんて無理な話ですが，“こんなことができて，こんなことはできない” といったことを大まかに理解しておくだけでも実際にプログラムを書く際に大きく違ってくるはずです．どんな演算機能が用意されているのかということだけでも押さえておくとよいでしょう．

5.5.12 スカラ変数を引数に取る型総称組み込み関数

これまで紹介した型総称組み込み関数の中には，spu_add や spu_or などのように，一部の引数にベクタ変数だけでなくスカラの変数やリテラルを取ることが可能なものが用意されています．このような型総称組み込み関数の場合，引数にスカラ変数を指定すると，演算命令の前に spu_shuffle 相当の処理が挿入されて，スカラ変数の値をベクタの全要素にコピーする命令が入ってしまう可能性があります．

たとえばリスト 5.17 の add_simple のように，コンパイルの時点では値が定まらないスカラ変数が渡されるような場合，add_complex と同じ命令列が生成されてしまいます．この場合，b がリテラル (定数) だったら 1 命令で済むところが，pattern のロードと spu_shuffle に相当する命令が加わるため，3 命令が生成されてしまうことになります．些細な違いではありますが，意識しておくと後々最適化を進める上で役立つことがあるかもしれません．

リスト 5.17 引数にスカラ変数を渡すのと同じ命令を生成する記述

```
vec_uint4
add_simple(vec_uint4 a, unsigned b)
{
    return spu_add(a, b);
}

vec_uint4
add_complex(vec_uint4 a, unsigned b)
{
    vec_uint4 b2 = spu_promote(b, 0);
    vec_uchar16 pattern = (vec_uchar16){0, 1, 2, 3, 0, 1, 2, 3,
                                        0, 1, 2, 3, 0, 1, 2, 3};
    b2 = (vec_uint4)spu_shuffle(b2, b2, pattern);
    return spu_add(a, b2);
}
```

5.6 SPU のパイプライン

続いて SPU の演算処理と関わりの深いパイプラインについて紹介します．SPU の命令処理を行う部分は，内部に奇数パイプと偶数パイプと呼ばれる非対称な構造の 2 本のパイプラインを持っています．命令の種類ごとにどちらのパイプで実行されるかが決まっており，5.4 で紹介し

[*14] この他にも型総称組み込み関数は色々とあるのですが，これらは演算とは関係がないので，また別の機会に紹介します．

た組み込み関数に対応する SPU の各命令は，2 本のパイプラインのいずれかで実行されます．

ここで SPU の命令について確認しておきましょう．SPU の命令は 1 命令あたり 32bit 幅の固定長です．したがって，SPU の実行ファイルには 32bit 幅の命令コードが大量に並ぶことになります．その際，**8 で割り切れるアドレス上に偶数パイプで実行される命令が配置されていて，さらにその直後の 8 で割ると 4 余るアドレスに奇数パイプで実行される命令が配置されていると，並んだ 2 つの命令が同時にパイプラインに投入されて並列に処理されます**．すなわち，1 サイクルでの 2 命令同時発行 (Dual Issue) が可能になります．

2 命令同時発行となるような命令の並べ替えなどの調整は基本的にコンパイラが行ってくれるのですが，プログラムの記述によってこれを制御することも可能です．これについての具体的な例は後ほど第 6 章にて紹介するとして，ひとまずここでは 5.4 で紹介した組み込み関数で利用可能な SIMD 演算がどちらのパイプで実行されるかを把握しておきましょう．大まかにまとめたものを表 5.4 に示します．逆数の近似値を導出する型総称組み込み関数は，内部で 2 つの命令に変換され，それぞれが奇数パイプと偶数パイプで処理されるため，表 5.4 では両方使うものとしています．また，表に記載していませんが，演算以外の重要な命令であるロード・ストア命令と分岐命令は奇数パイプで処理されます．その他の制御系の命令も奇数パイプで処理されます．

表 5.4　演算向けの処理とパイプの関係

偶数パイプを使うもの	加減算，乗算，キャスト， ビット単位の論理演算 (`spu_orx` 除く)， 比較処理，バイト操作， シフト・ローテート (非 `qw` 系)， `spu_cntb`，`spu_cntlz`，`spu_sel`
奇数パイプを使うもの	`spu_gather`，`spu_orx`，`spu_shuffle`， マスク生成，シフト・ローテート (`qw` 系)
両方使うもの	逆数の近似値導出

表を見ると，よく使いそうな演算は偶数パイプに集中しており，比較的イロモノの雰囲気漂う一部の演算処理のみが奇数パイプに割り振られている状況となっていることがわかるかと思います．そのため，LS の読み書きをあまり伴わない，レジスタ間で値を渡してひたすら演算を続けるような処理を SPU で行う場合，偶数パイプにばかり処理が偏り，奇数パイプは何もしていないという状態が発生してしまいます．このような場合，偶数パイプでやっている処理の一部を奇数パイプに移すことで SPE 全体の処理効率を上げることができます．この辺りに関する最適化手法については 6.6 にて紹介します．

5.7 SPE の浮動小数点数

SPE の演算について扱ったついでに，SPE での浮動小数点数における特筆すべき点についても触れておきたいと思います．ご存知の方も多いかもしれませんが，Cell/B.E. の SPE の浮動小数点数まわりの実装は，標準規格 (IEEE 754) から若干逸脱しています．これは，Cell/B.E. が主にゲームやマルチメディア演算を想定して開発されたためのようです．規格にきっちり準拠している他のプロセッサと同じノリで利用すると思わぬ落とし穴に落ちる可能性があるので，主

だった違いについて簡単に紹介しておきます．もっとも，筆者は IEEE 754 にさほど詳しいわけではないので，理解できる範囲での説明となってしまう点はご了承ください．本格的にきっちり知りたい場合は文献 [2] や文献 [3] を参照してください．

5.7.1　SPE の単精度浮動小数点数

まず，SPE の単精度浮動小数点数 (`float`) について，一般的な実装と異なる部分を紹介します．Cell/B.E. の SPU の単精度浮動小数点数は，IEEE 754 とフォーマットは同じなので，正規化数の範囲であればデータに互換性があるのですが，非正規化数や NaN，無限大といった特殊な数の扱いなどにおいて色々と違いがあります．

- NaN および無限大をサポートしていない

  IEEE 754 では，単精度浮動小数点数 32bit のビット 1 からビット 8 までの 8bit で指数を表現します．この部分が 255 になると，仮数の値に応じて NaN や無限大という扱いとなり，通常の数と扱いが変わります．しかしながら，Cell/B.E. の SPU ではこれらをサポートしていません．SPU は，指数部が 255 のときも単に指数が +128 の値として扱います．そのため，`float` 型のデータの演算可能な最大値が IEEE 754 と比べて倍になっています．また，それを超えるような結果となる演算が行われた場合，無限大がないため，結果として常に最大値が返ります．

- 非正規化数をサポートしていない

  SPU では，非正規化数 (指数部の 8bit が 0 で埋まった，0 に非常に近いことを示す値) を演算対象とする演算において，非正規化数は 0 として扱われます．また，IEEE 754 準拠の環境であれば演算結果が非正規化数となるような演算でも，SPU では結果が 0 になります．たとえばリスト 5.18 のような演算を行うと，SPE の場合，`f` は 0 になってしまいます．

リスト 5.18　SPU だと結果が非正規化数にならない演算の例

```
#include <float.h>
...
    volatile float f;
    f = FLT_MIN; // float型の正の正規化数の最小の値
    f /= 10;     // 10で割ると非正規化数にならずに0になる
    f *= 100;    // 当然100倍しても0のまま (PPEだとFLT_MINの約10倍の値になる)
```

- 丸めがゼロ方向丸め (切り捨て) にしか対応していない

  これが一番気になるという人も多いのではないかと思うのですが，SPU の単精度浮動小数点数は，誤差の丸めモードとしてゼロ方向丸め (切り捨て) にしか対応していません．Intel 系のプロセッサなどでプログラムを動かした場合，最近接値への丸めが標準となっているため，誤差の蓄積の仕方に違いが生じます．

- 演算結果に -0 (マイナスゼロ) が出ない

  SPU でも符号ビットが 1 で残りのビットが全て 0 の数は通常通り -0 として扱われますが，演算結果として -0 が出ることはありません．たとえば，-0 と -0 を足し算した場合，

Intel 系のプロセッサでは −0 が維持されるのに対し，SPU では 0 になります．

- 例外の検出

  Cell/B.E. の SPU では演算の過程で生じた以下の状態を検出できるようになっています．いずれも FPSCR という特別なユーザレジスタにフラグが立ち，これを読むことで何が起きたのかを知ることができます (ただし例外をトラップさせることはできない)．FPSCR の読み方については後述します．

  – オーバーフロー・アンダーフロー

    演算結果がオーバーフローした場合 (表現可能な最大値・最小値を超えた場合) やアンダーフローした場合 (表現可能な最も 0 に近い値よりも小さくなった場合) に発生

  – ゼロ除算

    割り算 (`spu_re` もしくは `spu_rsqrte` で実行される命令) で 0 で割ろうとした場合に発生

  – IEEE 非準拠結果

    演算などの結果が IEEE 754 の結果と異なる可能性がある場合に発生

なかなか制約が多く，処理の内容によっては実用的な結果が得られなくなる場合もあるかもしれません．この辺りは他の環境との相違を織り込んで割り切って使うなどの工夫が必要でしょう．

5.7.2 SPE の倍精度浮動小数点数

次に SPE の倍精度浮動小数点数 (`double`) の扱いについても紹介しておきます．Cell/B.E. の SPE の倍精度浮動小数点数は，IEEE 754 のデータフォーマットに準拠しているだけでなく，IEEE 754 の 4 種類の丸めモード (最近接値への丸め，ゼロ方向丸め (切り捨て)，$-\infty$ 方向への丸め，$+\infty$ 方向への丸め) にもきちんと対応しています．

しかしながら，こちらも完全に IEEE 754 準拠というわけではありません．たとえば，演算の結果 NaN が生じたとしても例外を発生させることはできず，常に qNaN として処理されて演算結果に蓄積されていきます．また，非正規化数が結果として得られることはあっても，オペランドとして利用する場合は 0 か −0 として扱われます．そして qNaN にしか対応しないという点と関連しますが，例外発生時にトラップする機能が搭載されていません．そのため，変な値が演算されそうになった際に，未然にそれを防いでソフトウェアで代行することができません．もちろん，例外が発生した場合は FPSCR のビットにフラグが立つので，その値を見て例外の有無を知ることは可能です．

また，SPU の倍精度浮動小数点数の演算命令は，オペランドのレジスタを `vec_double2` 型とみなすため，外から見ると 2 組の倍精度浮動小数点数の演算を同時に行っているように見えるのですが，実は SPU 内部では SIMD 処理されない実装となっています．`vec_double2` 型以外の型のベクタに対する演算では，SPU 内にベクタの要素数分の演算器が用意されているため，実際に並列に演算を行って結果を返すのですが，倍精度浮動小数点数に対応した演算器は SPU 内部には 1 個しかないようで，ベクタにパックされた 2 つの倍精度浮動小数点数のデータは分解さ

れて 1 データずつ処理されます．また，倍精度浮動小数点数の演算は処理に 13 サイクルかかりますが，そのうちパイプライン化されているのは最後の 7 サイクルのみで，倍精度浮動小数点数の演算を開始した直後の 6 サイクルはパイプラインがストールします．さらに，倍精度浮動小数点数の演算は偶数パイプで行われるのですが，奇数パイプの処理と組み合わせて 2 命令同時発行を行うことができません．

このように，倍精度の浮動小数点数に関しても色々と残念なところの多い Cell/B.E. の SPU なのですが，この辺りの状況は同じ CBEA 準拠のプロセッサでも実装によって異なるようです．筆者は触ったことがないのですが，IBM の提供している CBEA 準拠のプロセッサである PowerXCell 8i は，Cell/B.E. と異なり，この辺りが改善されているという話です．

5.7.3　FPSCR へのアクセスの仕方

浮動小数点数の演算の各種状態を保持するレジスタである FPSCR は，型総称組み込み関数 `spu_mffpscr` を用いて `vec_uint4` 型のベクタに読み出すことができます．また，`spu_mtfpscr` を用いることで，FPSCR に `vec_uint4` 型のベクタの値を書き込むことができます．

```
d = spu_mffpscr();
spu_mtfpscr(a);
```

FPSCR のビットフィールドを図 5.24 に，個々のビットフィールドのフラグの詳細を表 5.5，表 5.6，表 5.7 および表 5.8 に示します．なお，`vec_double2` 型のベクタに対する除算を行う演算はハードウェアでサポートされていないため，これに対するゼロ除算例外の情報は用意されていません．また，フラグにおいて "–" と表示されている部分は右側に示す状態の影響を受けないビットであることを示しています．

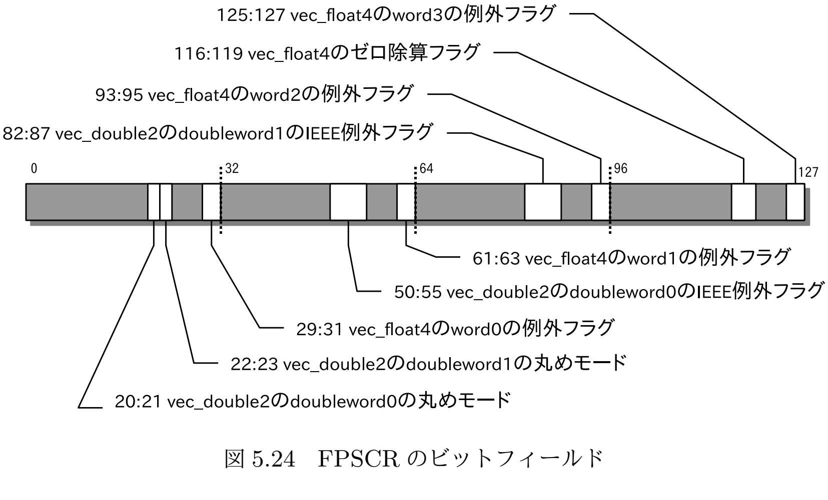

図 5.24　FPSCR のビットフィールド

表 5.5 vec_float4 型のベクタの演算における例外フラグ

フラグ	状態
1--	オーバーフロー発生
-1-	アンダーフロー発生
--1	IEEE 非準拠結果の出現の可能性

表 5.6 vec_double2 型のベクタの演算における例外フラグ

フラグ	状態
1-----	オーバーフロー発生
-1----	アンダーフロー発生
--1---	不正確な結果の出現
---1--	無効演算発生
----1-	qNaN が伝播している
-----1	非正規化数オペランドによる IEEE 非準拠結果の出現

表 5.7 vec_double2 型のベクタの各要素の丸めモード

フラグ	モード
00	最近接値への丸め
01	ゼロ方向丸め
10	$+\infty$ 方向への丸め
11	$-\infty$ 方向への丸め

表 5.8 vec_float4 型のベクタの各要素のゼロ除算フラグ

フラグ	状態
1---	word0 でのゼロ除算発生
-1--	word1 でのゼロ除算発生
--1-	word2 でのゼロ除算発生
---1	word3 でのゼロ除算発生

5.8 SIMD 数学ライブラリ

SIMD 数学ライブラリは，Cell SDK に付属しているライブラリで，SPU がハードウェアで提供している演算機能を組み合わせた，複雑で高度な数学系の関数の SIMD 演算を提供します．C99 で規定されている数学関数を SIMD 化したものが提供されている (拡張倍精度を用いるものなど一部を除く) 他，整数や浮動小数点数のベクタ同士の割り算などの関数もいくつか用意されています．

SIMD 数学ライブラリの API に関しては文献 [6] に記載されています．SPE で外部のライブラリを使用する場合，関数の呼び出しによるプログラムの実行速度への影響が気になりますが，SIMD 数学ライブラリでは，通常の静的ライブラリ以外に，中身のソースがまるごと入ったヘッダファイルが別途提供されているため，各関数の実体の記述を含むヘッダファイルを include して，API ではなく実体 (API 名の先頭に “_” をつけたもの) を呼び出すことで，その部分をインライン展開して，呼び出しコストを一切かけずに利用することが可能となります[*15]．

[*15] “一体どのようにしてこの演算を SPE で実現しているのか” ということをヘッダを読んで学ぶことも可能です．

また，ヘッダファイルとはいえ中身は関数の実装なので，プログラムに組み込んだ場合のライセンスが気になるかもしれませんが，SIMD 数学ライブラリは修正 BSD ライセンスでの配布となっているため，組み込んだことによってソースの開示義務が生じることはありません．

なお，SIMD 数学ライブラリでも，中では SPU のハードウェアの演算器を用いているので，5.7 で紹介した単精度浮動小数点数関連の規格の逸脱などはそのまま引きずることになります．たとえば SPU では単精度浮動小数点数の NaN や無限大がサポートされていませんが，無限大かどうかをテストする `isinf` の SIMD 版である `isinff4` のソースを見ると，中では `spu_splats` でリテラルの 0 を複製したものを返す (つまり引数によらず常に 0 が返る) 実装となっています．また，NaN を生成する `nan` に相当する関数はありません．丸め誤差などに関しても同様です．

また，SIMD 数学ライブラリで提供されている関数のうち，半分くらいのものは PPE でも同じ名前で利用可能になっています (倍精度浮動小数点数に対する SIMD 演算の関数は PPE にはない) が，上記のような理由から，同じ関数名で呼び出せる SIMD 演算でもハードウェアの実装の違いにより PPU 上で利用した場合と SPU 上で利用した場合とで演算結果が違ってくる可能性があります．PPE も交えて演算を行う場合などには注意が必要でしょう．

非常に簡単ではありますが，SIMD 数学ライブラリを `include` して使う例をリスト 5.19 に示します．リスト 5.19 では，SIMD 数学ライブラリの提供する `divf4` という単精度浮動小数点数のベクタ同士の割り算を用いて，全要素が 1 のベクタを割って逆数を求めることを行っています．`divf4` は中でニュートン・ラプソン法を使って精度を上げることをしているため，単純に `spu_re` を用いて逆数の近似値を求めただけの場合に比べて精度のよい値が求まります．リスト 5.19 では，`simdmath/divf4.h` を `include` し，`_divf4` を呼び出すことで演算を行っています (おまけとして `spu_re` も実行)．

リスト 5.19　SIMD 数学ライブラリを利用した除算

```
#include <stdio.h>
#include <spu_intrinsics.h>
#include <simdmath/divf4.h>
...
    vec_float4 funit = (vec_float4){1.0, 1.0, 1.0, 1.0};
    vec_float4 f1 = (vec_float4){1.0, 2.0, 3.0, 4.0}, f2, f3;

    // 1を各要素で割ることで逆数を計算
    // divf4は「_divf4」という名前で呼ぶことでインライン展開される
    f2 = _divf4(funit, f1);
    // 参考までにspu_re単体での逆数も計算
    f3 = spu_re(f1);
```

なお，リスト 5.19 を実行して，`_divf4` を利用して求めた逆数 `f2` と `spu_re` だけで求めた逆数 `f3` の各要素を出力すると，以下のようになります．`spu_re` の精度が非常に悪いことが見てわかるかと思います．本節の本題から外れますが，`spu_re` を直接使用する場合は精度に関して十分に注意する必要があると言えるでしょう．

```
_divf4: 1.000000 0.500000 0.333333 0.250000
spu_re: 0.999878 0.499939 0.333313 0.249969
```

第 6 章
SPE のプログラムの最適化

> おれが短い人生で学んだことは…………
> 人間は策を弄すれば弄するほど予期せぬ事態で策がくずれさるってことだ！…………
> 人間を超えるものにならねばな………
>
> — 荒木飛呂彦『ジョジョの奇妙な冒険』より —

本章では SPE のプログラムの最適化手法について，具体的な例を挙げつつ紹介していきます．

6.1 SPE のプログラムの最適化とは

第 2 章でも述べましたが，SPE は実行コードを動的に最適化して高速に動作させるようなハードウェアアーキテクチャを備えていません．そのため，SPE 上でのプログラムの実行性能は，実行コードがどれだけ静的に最適化されているかによって決まります．しかしながら，SPE 用の GCC である spu-gcc は，正確な SPE 用の実行コードを生成してくれるとは言っても，ベースとなっているのは汎用的なコンパイラである GCC です．最適化オプションを付加することでそれなりに効率のよい実行コードを生成してくれはしますが，ものすごく賢いというわけでもないので，適当に書いたプログラムはやはり適当なりの性能しか出てくれません[*1]．よって，SPE 上でとにかく高速に処理を行いたいのであれば，spu-gcc が効率的な実行コードを生成してくれるように人間がプログラミングの段階で色々と頑張って工夫をする必要があります[*2]．本章では，こういった人間が手と頭を使って頑張る最適化を中心に紹介していきます[*3]．

本章で紹介する最適化手法の多くは，SPE のアーキテクチャと spu-gcc のクセに基づくものになります．もちろん，ループの最適化やインライン展開の制御などに関しては他のアーキテクチャにも応用できますが，ここでは一般的にどうであるといったことよりも，“とにかく SPE はこうだからこうすれば速くなる” という点を中心に解説を行います．

また，本章で紹介する手法は，必ずしも “これをやればうまくいく” というものではありません．手法とあわせて紹介する具体例ではうまくいっているものでも，他のプログラムでも同様に性能向上につながるとは限りません．場合によってはかえって性能が悪化する場合もあります．

[*1] 一応ある程度自動で SIMD 化してくれる機能はあるらしいのですが，条件はかなり限られるようです．

[*2] 高機能な商用コンパイラを用いた場合，また話は違ってくるのかもしれませんが．

[*3] “このオプションをつけてコンパイルするとプログラムによっては最大 n 倍高速化します” なんてつまらないですよね ….

過信は禁物です.

6.1.1 最適化のサンプル

本章では，各種最適化の適用と効果の評価で用いる具体的な例として，2008 年 11 月頃から 2009 年 3 月頃までの期間に Fixstars 社主催で開催されていた Hack the Cell 2009 [*4] というプログラミングコンテストの課題を利用します.

Hack the Cell 2009 の課題は，メルセンヌ・ツイスタと呼ばれる疑似乱数生成アルゴリズムで生成される乱数系列の，ある範囲の値を足し合わせた結果を求めるのに要する時間を競うというものでした (乱数系列そのものを求める必要はない)．本章では，最適化手法を順番に紹介しつつ，それをこの課題で実行時間を競うのに用いた関数に適用して最適化していきます．また，その際，課題のプログラムに付属している実行時間計測機能を利用して，実際に最適化によってどれくらい速くなったのかを数値で示します．実行時間は条件によってあまり変わるものでもないので，サンプルとして課題に付属している 10 パターン (初期値や範囲などのパターン) のうちの最初のパターンを用います[*5]．また，SPE の特性上，DMA 転送などを途中に伴わないのであればプログラムの実行時間はほとんど変化しないことから，複数回測定を行って平均を取るといったことは行っていません.

本章で紹介する最適化の例を実際に手で入力しながら確かめていきたい場合は，コンテストの公式サイト[*6] で公開されている課題のソースコードをダウンロードして利用してください．なお，上記 URL のサイトからダウンロードできるソースコードでは，最適化対象となる関数 (`genrand_mine`) の中身は空っぽとなっています．具体的なソースコードについては後ほど紹介しますが，本章ではそこに最適化を施していないオリジナルのメルセンヌ・ツイスタのソースコードをほぼそのままはめ込んだものを最適化のスタートラインとします.

6.2 C 言語プログラマのためのアセンブリ言語超入門

第 1 章でも述べましたが，本書は C 言語による SPE プログラミングを扱った本です．本書ではインラインアセンブラも含めアセンブリ言語によるプログラムの記述は一切扱いませんので，本章で紹介するようなプログラムの最適化を行う場合でも，SPE のアセンブリ言語でプログラムが書ける必要はまったくありません．が，しかし，SPE のプログラムを最適化していくにあたり，SPE の実行コードの逆アセンブル結果が読めると何かと便利です．そこで，具体的な最適化のテクニックについて紹介する前に，最適を行う上で覚えておきたい実行コードの逆アセンブルの方法と，その結果の読み方 (というより “眺め方”) についてごく簡単に紹介しておきます.

[*4] `http://cell.fixstars.com/challenge/`

[*5] ソートなどと違い，問題のパラメータによって実行時間に大きな傾向の差が出たりはしないので，どれを使っても最適化具合を見る上で大きな違いはありません.

[*6] `http://cell.fixstars.com/challenge/challenge.html`

6.2.1 SPE プログラムの逆アセンブルの方法

コンパイラによって，どのような命令の並びが実行コードとして生成されたかを逆アセンブルして確認するには，`spu-objdump` コマンドを用います．以下のように `spu-objdump` コマンドに `-d` オプションをつけて SPE 用の実行ファイルなどを引数として渡すと，そのバイナリファイルの中の実行コードを逆アセンブルした結果を得ることができます．

```
% spu-objdump -d hello_spe

hello_spe:     file format elf32-spu


Disassembly of section .init:

00000000 <_init-0x4>:
   0:   40 20 00 00     nop     $0

00000004 <_init>:
   4:   24 00 40 80     stqd    $0,16($1)
...
00000160 <main>:
 160:   42 03 b8 03     ila     $3,1904 # 770
 164:   24 00 40 80     stqd    $0,16($1)
 168:   40 20 00 7f     nop     $127
 16c:   24 ff 80 81     stqd    $1,-32($1)
 170:   1c f8 00 81     ai      $1,$1,-32
 174:   33 00 11 80     brsl    $0,200 <puts>   # 200
 178:   40 80 00 03     il      $3,0
```

6.2.2 逆アセンブル結果の読み方

本書を読み進めるにあたり，SPE のアセンブリ言語の文法や命令表記を完全に理解して覚える必要などまったくありませんが，最適化したいところ (あるいは最適化されるように C 言語レベルで工夫をして記述したところ) に対して実際にどのような命令が使われていて，どのように処理が進んでいるのかを何となくでも読み取ることができるようになると，最適化の効率が大きく違ってきます．ここでごく簡単に読み方を紹介しておきます．

まず，目的の処理がどのように最適化されたかを見るには，その処理を含む関数を探します．C 言語でプログラミングしているのであれば，関数がインライン展開でもされたりしていない限り，“<” と “>” で囲われた関数名がどこかにあるはずです．それを探すところから始めましょう．たとえば先ほど示した逆アセンブル結果の例に “`<main>`” という記述がありますが，ここが `main` 関数の始まりに対応します．

中身を確認したい関数名が見つかったら，次は中の処理を見ていきます．関数の中の処理は，(当然ですが) SPE が直接処理する命令の形で記述されています．1 行につき 1 命令が記述されており，各行には命令のアドレス，命令を 16 進数表記したもの (いわゆる機械語) および命令を人間が読めるアセンブリ言語の形で表記したものが併記されています．

これらの中で，SPE の最適化の結果を確認する上で重要なのは，アセンブリ言語の形で表記

された命令の部分です．先に示した例では，“`ai $1,$1,-32`” などの部分がアセンブリ言語で表記された命令になります．左側にある “`ai`” や “`stqd`” などが命令名で，“ニーモニック” と呼ばれます．また，その右側にあるカンマ区切りの文字列の各々は “オペランド” と呼ばれ，命令で用いる値や結果の格納先を指定する部分になります．

オペランドの中の “`$`” で始まる数字はレジスタを表し，`$`の後の数字がレジスタ番号になります．括弧つきのレジスタ表記は，“括弧の中のレジスタに格納された値をアドレスとしてみなし，そのアドレスにアクセスする” ということを意味します．レジスタを変数ではなくポインタとして扱い，そこに “`*`” をつけてアクセスするようなイメージです．また，括弧の手前に書かれている数値は，括弧の中のレジスタの値をアドレスとみなしてアクセスする際に，アドレスにオフセットとして加える値になります．たとえば “`-32($1)`” とあったら，レジスタ`$1` の値から 32 を引いた値をアドレスとして扱います．一方，`$`がついていない数値は定数です．一部の命令はオペランドに直接値を記述できるようになっており，このような値を “即値” と呼んだりします．この他，オペランドの部分に “`<puts>`” のような “`<`” と “`>`” で囲われた文字列や，それに定数を足し算している表記が現れる場合がありますが，これらは分岐先の関数名や，その関数の先頭からのオフセット (関数の途中に飛ぶ場合) を表しています．オペランドの数や種類はニーモニックによって異なります．より詳しく知りたい場合は文献 [3] を参照してください．

非常に適当ですが，大体これくらい把握しておけば C 言語のコードがどのような感じの処理に変換されたのかを何となく雰囲気でつかめるかと思います．各命令の機能やオペランドの数・扱いなどに関しては，文献 [3] にすべて書かれていますので，そこまで知る必要が生じた場合はそちらを参照してください．

本章で最初に行う最適化である SIMD 化では，SPE の演算命令を狙って呼び出す型総称組み込み関数を用いるため，逆アセンブルして実行コードにどのような命令が使われるようになったのかを確認する意味はあまりありませんが，分岐やループの最適化においては，コンパイラがどのように判断してどのような実行コードを生成するのか，C 言語のレベルでは予測しにくい面があるため，逆アセンブルを行って，実際にどのような実行コードが生成されているのかを確認することが度々あります．アセンブリ言語だからと言ってあまり怖がらずに[*7]，是非眺めることに慣れておきましょう．

6.3 SIMD 化

ここからは最適化手法を順次紹介していきます．最初に SPE プログラムの SIMD 化による処理最適化を紹介します．SPE のプログラムの SIMD 化に必要となる SPE の持つ演算処理や，型総称組み込み関数を利用したそれらの呼び出し方などについては第 5 章で紹介済みなので，ここでは実際にプログラムを SIMD 化する際の流れを中心に紹介していきます．

[*7] そんなのは筆者だけかもしれませんが ···．

6.3.1 最適化前のソースコード

まず，SIMD 化を始める前に，SIMD 化の対象となるソースコードをリスト 6.1 に示します. 6.1.1 で述べたように，このソースコードは Hack the Cell 2009 の課題プログラムのテンプレートに，オリジナルのメルセンヌ・ツイスタのソースコードをほぼそのままはめ込み，課題で不要とされている処理を除去して少しだけ記述を手直ししたものになります*8．なお，関数だけでなく，関数内で利用している定数および関数外で確保されている変数や配列に関しても，プログラムの挙動を理解する上で必要なことから，あわせて記載しています.

リスト 6.1 最適化対象のソースコード

```
#define N 624
#define M 397
#define MATRIX_A 0x9908b0dfU
#define UPPER_MASK 0x80000000U
#define LOWER_MASK 0x7fffffffU

static unsigned mt[N] __attribute__((aligned(16)));
static int mti = N + 1;
...
unsigned
genrand_mine(int num_rand)
{
    unsigned r = 0;
    int i;

    for (i = 0; i < num_rand; i++) {
        unsigned y;
        unsigned mag01[2] = {0x0U, MATRIX_A};

        if (mti >= N) {
            int kk;
            // 配列mtのアップデート処理
            for (kk = 0; kk < N - M; kk++) {
                y = (mt[kk] & UPPER_MASK) | (mt[kk + 1] & LOWER_MASK);
                mt[kk] = mt[kk + M] ^ (y >> 1) ^ mag01[y & 0x1U];
            }
            for (; kk < N - 1; kk++) {
                y = (mt[kk] & UPPER_MASK) | (mt[kk + 1] & LOWER_MASK);
                mt[kk] = mt[kk + (M - N)] ^ (y >> 1) ^ mag01[y & 0x1U];
            }
            y = (mt[N - 1] & UPPER_MASK) | (mt[0] & LOWER_MASK);
            mt[N - 1] = mt[M - 1] ^ (y >> 1) ^ mag01[y & 0x1U];

            mti = 0;
        }
        y = mt[mti++];

        // Tempering
        y ^= (y >> 11);
        y ^= (y << 7) & 0x9d2c5680U;
        y ^= (y << 15) & 0xefc60000U;
        y ^= (y >> 18);
        r += y;
    }

    return r;
}
```

*8 意味なく（?）`unsigned long` になっている部分を `unsigned` に変更したり，文字がくっつき過ぎなところを少しゆとりを持たせたりといった変更を行っています.

処理の数学的な意味まで踏み込みはしませんが[*9]，簡単にこのプログラムが何をしているか見てみましょう．

最適化対象である `genrand_mine` 関数は，`int` 型の引数を1つ取り，`unsigned` 型の値を返します．引数 `num_rand` は乱数を何個分求めて足し合わせるかを示し，戻り値は足し合わせた結果 (チェックサム) になります．課題によると，引数は必ず4の倍数となるそうなので，今回の最適化では "`num_rand` は必ず4の倍数である" ということを前提として進めていきます．

関数の中を見ると，大きな `for` ループの中に `if` 文があり，その中に小さな `for` ループが2個入った構造となっています．この外側の `for` ループで行っている処理が，メルセンヌ・ツイスタの乱数系列を求めて足し合わせていく処理になります．外側の `for` ループが1回ループするたびに32bitの乱数が1個求まり ("`Tempering`" と書かれた部分より後で計算している `y` に乱数が格納される)，これを `r` に足し込んでいくことでチェックサムを求めています．

また，関数の外で，`mt` という要素数624の `unsigned` 型のデータの配列が確保されていますが，この配列が乱数の元となるデータを管理するテーブルとなります．このテーブルは，外側の `for` ループの先頭で `mti` が `N` (624) 以上だった場合に全要素がアップデートされます．`if` 文の中の `for` ループなどの処理がアップデート処理になります．アップデート処理の最後で `mti` は0に戻され，その後，外側の `for` ループを1回ループするたびに1増えて，また再び624に達した際にアップデートが行われます．したがって，外側の `for` ループが624回ループするたびに `mt` のアップデートが1回行われることになります．

6.3.2 SIMD化前のプログラムの整理

プログラムの処理の流れが大まかに理解できたら，次はプログラムをいじってみます．とはいえ，いきなり闇雲にSIMD化を始めても大概はうまくいきません．今回のようなちょっと入り組んだプログラムは，先に少し記述を整理しておくと，見通しがよくなってSIMD化や他の最適化を適用しやすくなる場合があります．まずはSIMD化のことを考えつつコードの整理から始めてみます．

最初から全体を考えるのは大変なので，ひとまず624回に1回だけ実行される `if` 文の中身をSIMD化することを考えてみましょう．624回に1回しか実行されないような部分を最適化して意味があるのかと思われるかもしれませんが，`if` 文の中には `for` ループが2個もあり，それぞれで結構な量の演算をしていることを考えると，`if` 文の外の処理を624回行うのと同じくらいかかっているように思われます[*10]．

`if` 文の中の処理に着目すると，`for` ループの中で `unsigned` 型の `mt` を1個1個アップデートしているように見えるので，雰囲気的にこれを4個ずつまとめてアップデートするようにSIMD化すればよさそうです．しかしながら，ここで気をつけなければならない点があります．データの依存性です．`mt` を4個まとめて計算しようと思っても，たとえば2個目のデータを求める際に1個目のアップデート後の値が必要だったとすると，1個目のアップデートが終わってからで

[*9] そもそも筆者はその辺まったく理解していないので，踏み込みたくても踏み込めません．

[*10] 真面目にやるならきちんとカウンタなどを用いて各処理にどれくらいかかっているかを測定すべきですが，ここでは手を抜いて勘で進めます．

ないと 2 個目のアップデートができないことになるため，1 個目から 4 個目までの 4 個を一斉に演算して結果を求めるといったことができません．また，`mt` のアップデートは 2 つの `for` ループ (とループの外の処理) にまたがっており，前半の `for` ループでは 227 要素が，後半の `for` ループでは 396 要素がそれぞれアップデートされています．ずいぶん中途半端な数な上に，前半のループでアップデートする個数は 4 で割り切れないため，そのまま SIMD 化したら半端分が出てしまいそうです．この辺をすっきりさせるために，このごちゃごちゃとした記述を整理してみます．

先ほども述べましたが，`if` 文の中の前半の `for` ループでは，`N` が 624 で `M` が 397 であることから `kk` が 0 から 226 までの値で変化し，227 回ループを回ることがわかります．そしてループの中では `mt[kk]` をアップデートしていることから，このループでは `mt[0]` から `mt[226]` までをアップデートしていることがわかります．一方，後半のループでは，`kk` は前半のループを抜けた時点での値である 227 から 622 までの値を取り，396 回ループを回ることがわかります．後半のループでも前半のループと同様に `mt[kk]` をアップデートしていることから，`mt[227]` から `mt[622]` までをアップデートしていることになります．

ここで，これら 2 つのループのアップデートの際の処理を眺めてみましょう．数学的な意味は置いておくとして，演算子などの記述をよく見ると，変数などに適用している演算自体は前半のループと後半のループとでとまったく同じであり，両者では単に演算に用いている `mt` のインデックスの計算方法が違うだけであることがわかります．

そして，最後に後半のループの直後にあるおまけの 2 行に着目してみると，ここでは `mt[623]` をアップデートすることを行っており，ここでもまた前半のループや後半のループとまったく同じ演算を異なるインデックス値の `mt` の要素に対して行っていることがわかります．

これらのことから，この 2 つのループとおまけの 2 行は，演算で用いる `mt` の要素のインデックスの導出を工夫することで，単一のループにまとめられることがわかります．`if` 文の中身を単一のループにまとめたものをリスト 6.2 に示します．

リスト 6.2 mt のアップデート処理をループ 1 つにまとめたもの

```
if (mti >= N) {
    int kk;
    // 配列mtのアップデート処理
    for (kk = 0; kk < N; kk++) {
        unsigned mt1 = kk < N - 1 ? mt[kk + 1] : mt[0];
        unsigned mt2 = kk < N - M ? mt[kk + M] : mt[kk + M - N];
        y = (mt[kk] & UPPER_MASK) | (mt1 & LOWER_MASK);
        mt[kk] = mt2 ^ (y >> 1) ^ mag01[y & 0x1U];
    }
    mti = 0;
}
```

リスト 6.2 では，`mt1` と `mt2` という変数を新規に導入し，`kk` の値によってインデックス値を計算して，適切な `mt` の要素を代入しています．このようにすることで，2 つのループの境目が 4 で割り切れないために SIMD 化できないところが発生してしまうといった問題が回避できそうです．続いて，このループを元に，SIMD 化で問題となるようなデータの依存性が存在していないかを確認していきます．

6.3.3 SIMD 化におけるデータ依存の確認

リスト 6.2 で示したループで，mt のある要素をアップデートするには別のどの mt の要素が必要になるのかについて考えてみましょう．前節で説明したように，一斉に複数のデータをまとめて演算しようとした際に，その中の後ろの方のデータの演算に，前の方のデータの演算結果 (新しい値) を必要とするような依存関係がある場合，これらを同時に演算すると，前から 1 要素ずつ演算したときと異なる値となってしまいます．このような問題の有無を確認するために，mt の各要素がアップデートの際にどの要素の値に依存するのかを図にまとめてみました．図 6.1 に依存関係を示します．

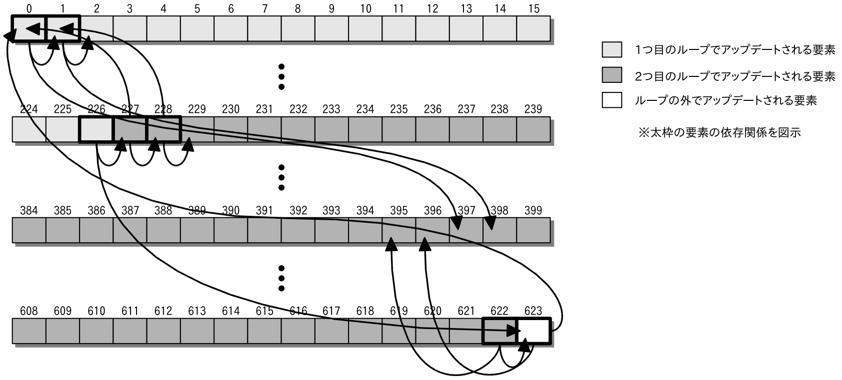

図 6.1 mt の依存関係

わかりにくいかもしれませんが，図のマス 1 個 1 個が mt の各要素となり，マスの上の番号が各要素のインデックス番号を表しています．また，薄いグレーで着色されたマスは前半のループでアップデートされる要素，濃いグレーで着色されたマスは後半のループでアップデートされる要素，そして最後の白抜きのマスはループの外でアップデートされる要素を表しています．矢印は依存関係を表しており，太枠で囲われた要素をアップデートするには，その要素から出ている矢印の指す先の要素の値が必要になるということを意味します．なお，図中のすべての要素について依存関係を示すと図がとても見辛くなってしまうので，依存はループの境界となる一部の要素 (図で太枠で囲われたマス) に関してのみ示しています．この図を元に，SIMD 化における依存関係について考えてみましょう．ここでは，配列の要素の新しい値 (アップデート後の値) を先頭から 4 要素ずつまとめて同時に演算できるかどうかについて考えてみます．

いきなり 4 要素まとめて考えるのは辛いので，まずは 2 要素ずつの組で考えてみます．ある要素と別の要素が同時に演算できるかどうかというのは，演算を始めようとした時点で，それら要素の演算に必要な値が揃っているかということに置き換えて考えることができます．ある要素 (要素 A) と別の要素 (要素 B) について考えた場合，元々要素 A より要素 B が後にアップデートされる関係であり，要素 A をアップデートしようとした時点で要素 B のアップデートに必要な値が既に確定している (要素 A をアップデートしてから要素 B をアップデートするまでの間に，要素 B が依存している値が変わることがない) ことが確実であれば，これら 2 つの要素は同時にアップデートできることになります．

たとえば mt[5] が mt[4] と同時に演算できるかどうか考えてみましょう．1 要素ずつ前から順番にアップデートしていったと考えた場合，mt[5] のアップデートに必要となる mt[6] のアップデート前の値と mt[402] のアップデート前の値は，mt[4] をアップデートするよりもずっと前に求まっており，mt[4] をアップデートしている間に値が変わるようなことはありません (mt[4] をアップデートしている間は mt[4] の値だけが書き換わるので)．したがって mt[5] は mt[4] と同時に演算ができるということになります．

同様に mt[4] と mt[6] について考えてみても，やはり同時に演算できることがわかり，最終的に一番遠いところで 226 要素先の mt[230] までが mt[4] と同時に演算できることがわかります (その次の mt[231] は，アップデート後の mt[4] の値に依存するため同時に演算できない)．また，mt[5] を基準に考えても，同様に mt[231] までの組み合わせで同時に演算できることがわかるので，先ほどの mt[4] の例と組み合わせて考えると，mt[4] から mt[6] までの 3 つの要素を同時に演算できるということがわかります．

さらに他のすべての要素について考えてみても，同様にその要素から 226 要素先までの要素とのペアで同時に演算できることがわかる (末尾まで到達した場合は配列の先頭に折り返して考える) ので，これらを総合すると，ある要素から 226 要素先の要素までの間のすべての要素を同時に演算できることになり，mt のどの要素から始めても，3 要素先までを含んだ 4 要素の同時演算は余裕で可能であるということが結論づけられます．

なお，依存関係の調査とは直接関係ありませんが，元のプログラムでループが 2 つに分かれていたのは，依存する要素が 1 個ずつずれていき，mt の末尾に到達してしまったために先頭に戻す必要があったためだということも図から読み取ることができます．これもループの特徴をつかむ上で非常に重要な事柄です．

6.3.4 ベクタへのロード

配列の値をベクタ変数に読み込む場合，5.3.1 で述べたように，16Byte 境界からデータを読み込む必要があります．ここで扱っている例で考えると，配列 mt が 16Byte 境界から始まっていれば (16Byte 境界にアラインしていれば)，mt[0] から順番に 4 要素ずつベクタ変数に読み込んで，SIMD 演算を行ってストアすることが可能となります．逆に，そのようになっていなかった場合，先頭の数要素を SIMD 演算せずに処理した上で，16Byte 境界から SIMD 演算を始めるような実装として対処します．

ここでリスト 6.1 をよく見ると，mt の直後に “__attribute__((aligned(16)))” という記述がついています．5.3.1 でも少し触れましたが，この記述があると，その手前で宣言した変数が 16Byte 境界にアラインした状態で確保されるようになります (状況によってはならない場合もあり)．この記述の詳細については別途 10.1.5 にて紹介しますが，ひとまずこの記述があることで mt が 16 の倍数のアドレスから始まることが保証されることから，mt[0] から mt[3] までの 4 要素をそのままベクタ変数にロードして演算を適用し，結果を格納したベクタ変数をそのまま同じ 4 要素にストアするような SIMD 演算が可能となります．

しかしながら，mt[0] から mt[3] までの 4 要素のアップデートには，それら 4 要素の値だけでなく，他の依存する要素の値も必要になります．そのため，4 つの要素をまとめて SIMD 演算

でアップデートするには，その 4 つの要素の値だけでなく，他の依存する要素についても同じようにベクタ変数にパックしておかなければなりません．そしてここで次の問題が生じます．

図 6.2 に，mt の先頭の 4 要素 (mt[0] から mt[3] まで) が自分自身以外のどの 4 要素に依存しているかをまとめてみました．図より，先頭の 4 要素が依存する要素は 16Byte 境界にアラインしておらず，そこから 4Byte 分ずれたアドレスから始まっていることがわかります．また，その後のすべての要素についても，4 要素ずつで考えた場合，依存する要素は同様に 16Byte 境界にアラインしていないことがわかります．5.3 で述べたように，LS からベクタにコピーできるのは 16Byte 境界にアラインされたデータに限られるため，このようにずれたところから始まっている 16Byte のデータを単一のベクタに直接格納することはできません．ではどのようにすればよいのでしょうか．

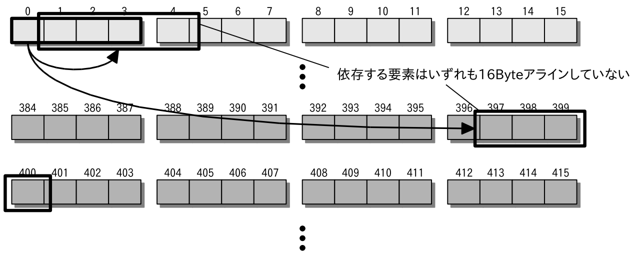

図 6.2　mt の先頭 4 要素が依存する要素の配置

このような場合，通常は図 6.3 に示すように，欲しい 16Byte のデータがまたがっている 2 つの 16Byte の領域のデータを一旦別々のベクタに読み出して，その後，それらから spu_shuffle で欲しいところのみを抽出して新たなベクタを作ることを行います．

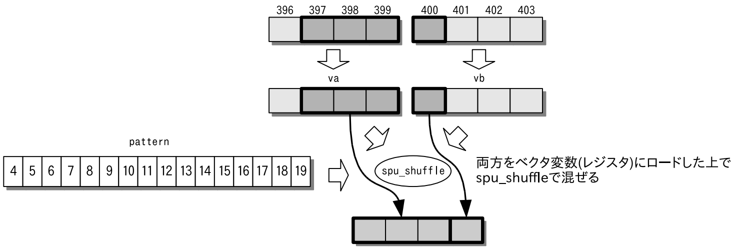

図 6.3　2 つのベクタをロードして spu_shuffle で混ぜて欲しいところだけ取り出す例

実際にやってみましょう．まず，mt の各要素を 4 要素まとめて演算しようとした場合に，それら 4 要素が依存する別の 4 要素がまたがっている，2 つの 16Byte 境界から始まる 16Byte の領域がどこになるのかを整理したものをリスト 6.3 に示します．4 で割って 1 を引いたりなどを考えるのがしんどいので，リスト 6.3 では定数である M や N をすべて展開しています．また，4 要素ずつまとめて処理するようにしたので，ループ回数が 1/4 となっています (ループ変数も見にくいので kk から k に変更)．vmt は mt を 16Byte 単位でアクセスするために別途定義したポインタです．なぜこのように記述できるのかについては，図 6.1 や図 6.2 を見て考えてみてください．とにかくこのように記述することで，一度に演算を行いたい 4 要素が依存している別の 4 要素が v1a，v1b，v2a および v2b の 4 つのベクタ変数にロードされることになります．

リスト 6.3 演算で必要となるベクタをロードする処理

```
vec_uint4 *vmt = (vec_uint4 *)mt;
vec_uint4 v1a, v1b, v2a, v2b;
...
        for (k = 0; k < 156; k++) {
            v1a = vmt[k];
            v1b = k < 155 ? vmt[k + 1] : vmt[0];
            v2a = k < 57 ? vmt[k + 99] : vmt[k - 57];
            v2b = k < 56 ? vmt[k + 100] : vmt[k - 56];
```

次に，v1a と v1b のペアおよび v2a と v2b のペアから spu_shuffle を用いて必要な部分を抽出してベクタを作ります．リスト 6.3 の処理にこの操作を加えたものをリスト 6.4 に示します．

v1a と v1b を混ぜたものがベクタ変数 v1 に格納され，v2a と v2b を混ぜたものがベクタ変数 v2 に格納されます．これらを混ぜる際には，パターンとして vshuf を利用しています．vshuf の値を見ると，spu_shuffle において，第 1 引数のベクタ変数の word1 から word3 と第 2 引数のベクタ変数の word0 とを連接したものを返すパターンになっていることがわかります．これで，mt のアップデートの SIMD 化において，4 要素をまとめて演算するのに必要となる 2 つのベクタ変数が用意できました．

リスト 6.4 演算で必要となるベクタをロードして混ぜ合わせる処理

```
// 一方のベクタのword1～word3ともう一方のベクタのword0を混ぜるパターン
const vec_uchar16 vshuf = (vec_uchar16){4, 5, 6, 7, 8, 9, 10, 11,
                                        12, 13, 14, 15, 16, 17, 18, 19};
vec_uint4 *vmt = (vec_uint4 *)mt;
vec_uint4 v1a, v1b, v2a, v2b;
vec_uint4 v1, v2;
...
            v1a = vmt[k];
            v1b = k < 155 ? vmt[k + 1] : vmt[0];
            v2a = k < 57 ? vmt[k + 99] : vmt[k - 57];
            v2b = k < 56 ? vmt[k + 100] : vmt[k - 56];
            v1 = spu_shuffle(v1a, v1b, vshuf);
            v2 = spu_shuffle(v2a, v2b, vshuf);
```

6.3.5 mt のアップデート処理の演算部分を SIMD 化する

ようやく mt のアップデート処理の演算部分の SIMD 化に入る準備が整いました．それではまず，リスト 6.5 の処理を SIMD 化することを考えてみましょう．

リスト 6.5 演算処理 1

```
            y = (mt[kk] & UPPER_MASK) | (mt1 & LOWER_MASK);
```

UPPER_MASK は 0x80000000U であり，LOWER_MASK は 0x7fffffffU であるとそれぞれ定義されているので，y は mt[kk] の MSB と，mt1 の LSB からの 31bit を混ぜたものになります．リスト 6.4 において v1 が mt1 を 4 要素分並べたものに相当するので，これを素直に SIMD 化するとリスト 6.6 のように書けます．

リスト 6.6 演算処理 1 を spu_and と spu_or で SIMD 化した例

```
vec_uint4 vy;
...
            vy = spu_or(spu_and(v1a, UPPER_MASK),
                        spu_and(v1, LOWER_MASK));
```

あるいは，この処理が 2 つのベクタをビット単位で取り出して混ぜる処理であることから，リスト 6.7 のように spu_sel を用いて SIMD 化することもできます．こちらの方が命令数が少ないのでおすすめです．

リスト 6.7 演算処理 1 を spu_sel で SIMD 化した例

```
const vec_uint4 vsel = spu_splats(0x7fffffffU);
vec_uint4 vy;
...
            vy = spu_sel(v1a, v1, vsel);
```

次にリスト 6.8 の記述を SIMD 化してみましょう．

リスト 6.8 演算処理 2

```
            mt[kk] = mt2 ^ (y >> 1) ^ mag01[y & 0x1U];
```

実際に mt の要素に新しい値を代入している部分です．新しい値は，離れたところにある要素 mt2 と，直前に求めた y を 1bit 右シフトした値および mag01 の要素を XOR することで求めています．mag01[y & 0x1U] の部分は，配列 mag01 を，直前に求めた y の LSB をインデックスとして参照するため，1 要素目か 2 要素目のどちらかになります．よって，この部分は y の LSB が 0 だったら 0x0U となり，1 だったら MATRIX_A (0x9908b0dfU) となります．この部分を SIMD 化した例をリスト 6.9 に示します．

リスト 6.9 演算処理 2 を SIMD 化した例

```
const vec_uint4 vone = spu_splats(0x1U);
vec_uint4 vmag01;
...
            vmag01 = spu_and(spu_cmpeq(spu_and(vy, vone), vone),
                             0x9908b0dfU);
            vmt[k] = spu_xor(spu_xor(v2, spu_rlmask(vy, -1)), vmag01);
```

この例では，先に元々のソースコードで配列 mag01 を参照した際に得られる値を 4 要素分一気に求めて vmag01 というベクタに格納しています．配列 mag01 は，y の LSB で 0x0U か 0x9908b0dfU か決まっていたので，ベクタの各ワードスロット内の LSB が 0 だったら 0x0U が入り，1 だったら 0x9908b0dfU が入るようなベクタの変換処理を作成すればよいことになります．ここではまず 4 要素分の y に相当する vy に対して，4 要素が全部 0x1U のベクタを spu_and で AND することで，その LSB のみを取り出し，その結果を 4 要素が全部 0x1U のベクタと spu_cmpeq で比較しています．これにより，vy の各要素の LSB が 0 だったら対応する要素に 0x0U が入り，1 だったら 0xffffffff が入ったベクタが得られます．後はそのベクタの各要素と 0x9908b0dfU の AND を取ることで，LSB が 0 だった要素に対応するスロットに 0x0U が入り，そうでないスロットには 0x9908b0dfU が入ったベクタを生成することができます．すなわち，mag01 を 4 回に分けて参照したのと等価な結果が得られることになります．

その次に，4 要素分のアップデート後の値を求めて元の配列に書き込むことを行っています．ここでは SIMD 化前の処理に忠実に，mt2 に相当する要素を 4 要素分まとめた v2 と，y に相当する要素が 4 要素分まとまっている vy の各要素を 1bit 右シフトしたものとを XOR し，その結果を直前に求めた vmag01 と XOR しています．これで 4 要素分を同時に計算して書き戻す処理

ができあがりました．

6.3.6 これまでのまとめ

ひとまずここまで個別に示してきた SIMD 化に関する記述をひとまとめにしてみます．vy の導出に spu_sel を用いたとすると，リスト 6.10 のようになります．だいぶ長くなりましたが，これで mt をアップデートする演算の SIMD 化はひとまず完了です．

リスト 6.10 mt のアップデート処理を SIMD 化した例

```
const vec_uint4 vone = spu_splats(0x1U);
const vec_uint4 vsel = spu_splats(0x7fffffffU);

const vec_uchar16 vshuf = (vec_uchar16){4, 5, 6, 7, 8, 9, 10, 11,
                                        12, 13, 14, 15, 16, 17, 18, 19};
vec_uint4 *vmt = (vec_uint4 *)mt;
vec_uint4 v1a, v1b, v2a, v2b;
vec_uint4 v1, v2;
vec_uint4 vy;
vec_uint4 vmag01;
...
    if (mti >= N) { /* generate N words at one time */
        int k;
        for (k = 0; k < 156; k++) {
            // vmtのアップデートに必要な要素をロード
            v1a = vmt[k];
            v1b = k < 155 ? vmt[k + 1] : vmt[0];
            v2a = k < 57 ? vmt[k + 99] : vmt[k - 57];
            v2b = k < 56 ? vmt[k + 100] : vmt[k - 56];
            // ロードした要素からvmtに対応したベクタ変数を作る
            v1 = spu_shuffle(v1a, v1b, vshuf);
            v2 = spu_shuffle(v2a, v2b, vshuf);
            // vmtのアップデート
            vy = spu_sel(v1a, v1, vsel);
            vmag01 = spu_and(spu_cmpeq(spu_and(vy, vone), vone),
                             0x9908b0dfU);
            vmt[k] = spu_xor(spu_xor(v2, spu_rlmask(vy, -1)), vmag01);
        }
        mti = 0;
    }
```

それではここで，ここまでの SIMD 化によってどれだけ処理時間が短縮されたかを確認してみましょう．genrand_mine が最初に示したリスト 6.1 の状態のものと，リスト 6.10 で示す SIMD 化を適用したものとで実行時間を比較した結果を表 6.1 に示します．6.1 で述べたように，実行時間は Hack the Cell 2009 の課題プログラムで測定し，1 パターン目の数値を用いています．実行時間の単位については後ほど 6.9 にて紹介しますが，ひとまずは相対的に数値がどれくらい変化したかについて見てください．全体を SIMD 化したわけでもないので，さすがに 4 倍とまではいきませんでしたが，まずは 1.5 倍ほど速くなっていることがわかります．

表 6.1 mt のアップデート処理の SIMD 化による性能向上

状態	実行時間
初期状態	171,645,615
アップデート処理 SIMD 化後	116,103,216

6.3.7 ループ全体の SIMD 化

せっかくなので，先ほど SIMD 化しなかった Tempering と呼ばれるリスト 6.11 の処理についても SIMD 化してみましょう．

リスト 6.11 Tempering の処理

```
y = mt[mti++];

// Tempering
y ^= (y >> 11);
y ^= (y << 7) & 0x9d2c5680U;
y ^= (y << 15) & 0xefc60000U;
y ^= (y >> 18);
r += y;
```

リスト 6.11 の処理は毎ループ `mt` から 1 要素，すなわち `unsigned` 型のデータを 1 個だけ読み出して，それにビットシフトと AND と XOR を何度か適用して，得られた結果 (乱数) を `r` に足し込む処理であるため，ループの中で複数要素分の演算を同時に行うことができません．4 つ同時に処理したくても，1 回のループで同時に処理できるデータが 1 つしかないのです．このような部分を SIMD 化するには，少し視野を広げて処理を見直す必要があります．

元々，外側の `for` ループの中で何をやっていたのかを考えてみると，624 回に 1 回だけ `mt` をアップデートし，そこでアップデートした `mt` の値を毎ループ 1 個ずつ前から順番に取り出して Tempering 処理をしていました．しかしながら，ここで `mt` を 624 回に 1 回の頻度で 624 要素まとめてアップデートすること自体には，効率面以外での意味はありません．624 ループかけて `mt` を前から 1 要素ずつ順番にアップデートしたとしても，Tempering の処理ではそれより後の要素を必要としないことから結果が変わることはありません．言葉だけではよくわからないかもしれないので，リスト 6.2 の記述を元に，1 回のループで 1 要素ずつアップデートするように変更した `genrand_mine` をリスト 6.12 に示します．

リスト 6.12 では，ループが 624 回繰り返されるたびに 1 回だけ `mt` の 624 要素をまとめてアップデートするのではなく，代わりに 624 回のループの各々で `mt` を 1 要素ずつアップデートするようになっています．

リスト 6.12 の形に変換すると，Tempering の SIMD 化の方針が見えてきます．`for` ループのループ回数を `num_rand` 回からその 1/4 の回数に減らし，その代わりループの中を SIMD 化して `mt` を 4 要素まとめてアップデートするようにした上で，さらにアップデート直後の 4 要素をそのまま用いて Tempering の処理も 4 要素分同時に行えば，全体が SIMD 化できます．

ここで気にしなければならないのが Tempering の処理での `y` の依存関係でしょう．`mt` のアップデートを SIMD 化する際に述べたのと同様に，もし，あるループで行う `y` の演算にその直前 3 回以内のループで求めた `y` の値が用いられていたとすると，ループ間で依存が存在することになるため，4 個分の `y` をまとめて演算することができなくなります．しかしながら，Tempering の処理を見ると，`y` は自身の値を定数を相手に演算しているだけであり，結果も `r` に足して終わりなので，その後の別のループで行われる Tempering 処理に一切影響しないことがわかります．`y` の値自体も，同じループ内で生成される `mt` の値しか利用していないため，`mt` の生成が SIMD 化できる限りは `y` も SIMD 化して 4 個分まとめて演算しても問題ないと言えます．

リスト 6.12 mt のアップデート処理をループ 1 つにまとめたもの

```
unsigned
genrand_mine(int num_rand)
    unsigned r = 0;
    int i;
    int kk;

    for (i = 0, kk = 0; i < num_rand; i++, kk++) {
        unsigned y;
        unsigned mag01[2] = {0x0U, MATRIX_A};

        if (kk == 624)
            kk = 0; // 624まで到達したら0に戻す

        unsigned mt1 = kk < N - 1 ? mt[kk + 1] : mt[0];
        unsigned mt2 = kk < N - M ? mt[kk + M] : mt[kk + M - N];
        y = (mt[kk] & UPPER_MASK) | (mt1 & LOWER_MASK);
        mt[kk] = mt2 ^ (y >> 1) ^ mag01[y & 0x1U];

        y = mt[kk];

        // Tempering
        y ^= (y >> 11);
        y ^= (y << 7) & 0x9d2c5680U;
        y ^= (y << 15) & 0xefc60000U;
        y ^= (y >> 18);
        r += y;
    }
    return r;
}
```

以上のことを踏まえて Tempering 処理も含めて SIMD 化を行ったものをリスト 6.13 に示します.

リスト 6.13 Tempering も含めて SIMD 化した例

```
    unsigned r = 0;
    int i;
    int k;
    const vec_uchar16 vshuf = (vec_uchar16){4, 5, 6, 7, 8, 9, 10, 11,
                                            12, 13, 14, 15, 16, 17, 18, 19};
    const vec_uint4 vone = spu_splats(0x1U);
    const vec_uint4 vsel = spu_splats(0x7fffffffU);
    vec_uint4 *vmt = (vec_uint4 *)mt;
    vec_uint4 v1a, v1b, v2a, v2b, v1, v2;
    vec_uint4 vmag01;
    vec_uint4 vsum = spu_splats(0U);
    vec_uint4 vy, vy2;

    for (i = 0, k = 0; i < num_rand / 4; i++, k++) {
        if (k == 156)
            k = 0; // 156回ループすると624要素処理したことになるので0に戻す

        v1a = vmt[k];
        v1b = k < 155 ? vmt[k + 1] : vmt[0];
        v2a = k < 57 ? vmt[k + 99] : vmt[k - 57];
        v2b = k < 56 ? vmt[k + 100] : vmt[k - 56];
        v1 = spu_shuffle(v1a, v1b, vshuf);
        v2 = spu_shuffle(v2a, v2b, vshuf);
        vy = spu_sel(v1a, v1, vsel);
        vmag01 = spu_and(spu_cmpeq(spu_and(vy, vone), vone), 0x9908b0dfU);
        vmt[k] = spu_xor(spu_xor(v2, spu_rlmask(vy, -1)), vmag01);

        // アップデートした直後の4要素に対してTempering処理を行う
        vy2 = vmt[k];
        vy2 = spu_xor(vy2, spu_rlmask(vy2, -11));
        vy2 = spu_xor(vy2, spu_and(spu_sl(vy2, 7), 0x9d2c5680U));
        vy2 = spu_xor(vy2, spu_and(spu_sl(vy2, 15), 0xefc60000U));
        vy2 = spu_xor(vy2, spu_rlmask(vy2, -18));
        vsum = spu_add(vsum, vy2); // 4要素ごとにバラバラに足し合わせていく
    }
```

```
    // 最後にvsumの4つの要素の和をword0に作る
    vsum = spu_add(vsum, spu_rlqwbyte(vsum, 8));
    vsum = spu_add(vsum, spu_rlqwbyte(vsum, 4));
    return spu_extract(vsum, 0);
```

元の記述から大きく変化したことに驚くかもしれませんが，よく見るとループの最初でやっている処理はリスト 6.10 と同じです．そして for ループの形もリスト 6.12 のループ回数が 1/4 になった以外に違いはありません．1/4 になっているのは，SIMD 化したことで，1 回のループで 4 個分の乱数を同時に計算できるようになったためです．

一方，新しく出てきた vy2 は，元のプログラムの y に相当するベクタです．vy2 のアップデート処理自体は，元々の Tempering の処理をそのまま SIMD 演算の型総称組み込み関数に置き換えただけなので，特筆するべきことはないでしょう．

また，演算が終わった後は unsigned 型の合計値保持用の変数に vy2 の各要素をその場で足し合わせることを行っていません (ベクタの各要素を取り出す処理は演算時間がかかるため)．代わりに，vsum というベクタ変数の各要素に vy2 の各要素を足し合わせています．これは，符号なしの整数の足し算が，足す順番を入れ替えても答えが変わらないという性質を利用したものになります．ループを抜けた後で vsum の 4 つの要素の和を求めれば，すべての乱数の和を求めたのと同じことになります．これを行っているのがループの直後にある 2 行です．自身と，自身を 8Byte ローテートさせたものとを足すことで，結果の word0 には元々の word0 と word2 を足した答えが入った状態となり，また word1 には元々の word1 と word3 を足した答えが入った状態となっています．これと，これをさらに 4Byte ローテートさせたものとを足し合わせることで，結果のすべてのワードに word0 から word3 までを足し合わせた数が入ることになるため，そのプリファードスロットの値を取り出して genrand_mine から return しています．

SIMD 化と端数処理

処理の SIMD 化を行うことで，今まで 1 個 1 個バラで扱ってきたデータを 4 個や 8 個といった大きな単位でまとめて扱うようになりますが，その際，まとめて扱うことのできない半端な数のデータ (端数) が発生してしまうことがあります．端数がそれ以外のデータに比べて極端に少ないのであれば，端数の処理が多少遅くても全体の性能にはほとんど影響しないのですが，1 回のループの最後に必ず端数処理が入ってしまうような場合などでは，端数処理のための特別な条件分岐やループが全体の性能に無視できないレベルの影響を与える可能性があります．ごくわずかな例外的なデータのために全体の処理効率が低下するのは非常にもったいないので，このような場合，極力端数処理を回避することを考えるべきです．端数処理の避け方は色々と考えられますが，たとえば処理するデータの末尾の半端な部分にパディングを付加して 4 の倍数にして，SIMD 演算ですべてを処理できるようにする方法が考えられます．その場合，パディングが演算に巻き込まれることになるので，パディングには全体の最終的な結果に影響しないような値を選んで設定する必要があるかもしれません．また，どうにも端数の発生を防ぐことができない場合でも，これを極小のループや分岐で対処せずに，“端数があってもなくても関係なく同じ命令を実行する” という漢らしいコードに切り替えることで乗り切る方法も考えられます．これについては 6.4 にて紹介します．

さて，この変更を加えてどれくらい速くなったか計測してみましょう．測定した結果を表 6.2 に示します．

表 6.2　処理全体の SIMD 化による性能向上

状態	実行時間
mt のアップデート処理のみ SIMD 化	116,103,216
Tempering 処理も含めて全体を SIMD 化	52,923,218

Tempering 部分を SIMD 化したことで倍以上高速化されました．SIMD 演算を用いて 4 組のデータを同時に演算するようにしただけでも相当な効果が得られたことがわかります．

SIMD 化による最適化はひとまずここでおしまいにしますが，このプログラムにはまだまだ最適化ポイントがたくさんあります．次節以降もこの例を用いて SIMD 化以外の最適化を順次適用して進めていきます．

6.4　分岐の最適化

SPE はハードウェアによる動的分岐予測の仕組みを備えていません．そのため，何も考えずに if 文などの分岐を多用したプログラムを書いてしまうと，分岐ミスによるパイプラインのフラッシュが頻発し，性能が大幅に悪化するおそれがあります．しかしながら，分岐によるダメージを軽減するようなプログラミングを心がければ，この問題はある程度回避することができ，何も考えずにプログラムを記述した場合に比べて性能を改善できる可能性があります．以下でそのためのテクニックをいくつか紹介します．

6.4.1　分岐ヒントを使う

通常，SPE では，命令を LS からアドレス順にどんどん先読みしてバッファに蓄えつつパイプラインに投入して実行していくので，分岐命令では分岐しない方がロスが少なくて済みます．よって，頻繁に実行されるような処理はなるべく分岐命令の直後に置いて分岐が不成立のときに実行されるようにし，逆に滅多に実行されないような処理は分岐成立時に実行されるように遠くに配置することで，プログラム実行時の分岐によるロスを低く抑えることができます．

しかしながら，コンパイラからしてみると，ある if 文の中の条件式が真になりがちなのか偽になりがちなのかの判断がつきません．そのため，何も考えずにプログラムを記述した場合，本来滅多に実行されることのない処理が分岐命令の直後に配置された実行コードが生成されてしまう可能性があります．

ちょうど，前節で扱った SIMD 化後のソースコードに，滅多に成立しない if 文があります．リスト 6.14 に示す部分です．この if 文の条件式は，ループが 156 回繰り返される間に 1 回しか真になりません．したがって，コンパイラがこの if 文をそのまま素直に記述通りに展開していたとすると，155/156 の確率で分岐ミスが発生することになります．

リスト 6.14　滅多に起こらない分岐の例

```
for (i = 0, k = 0; i < num_rand / 4; i++, k++) {
    if (k == 156)
        k = 0;
```

実際にどのような実行コードが生成されているのか，コンパイル結果を `spu-objdump` で逆アセンブルして見てみましょう．比較を行っている場所は "156" をキーワードに探すとすぐに見つかります．逆アセンブル結果の `if` 文に該当する処理をリスト 6.15 に示します．

リスト 6.15　滅多に起こらない分岐の部分を逆アセンブルした結果

```
4d4:    7c 27 02 13      ceqi     $19,$4,156       # 9c
...
4dc:    20 7f e9 93      brz      $19,428 <genrand_mine+0x60>      # 428
```

"`ceqi`" は，レジスタの各ワードスロットと即値 (命令に埋め込んだ数) の一致不一致を調べる命令です．`spu_cmpeq` で `vec_uint4` とリテラルを比較した場合に生成されます (この例では `if` 文の条件式で生成されている)．ここではレジスタ$4 が 156 かどうかを調べ，結果をレジスタ$20 に格納しています．一致した場合にレジスタ$20 の対応するワードスロットに `0xffffffffU` が格納され，そうでなかった場合は `0x0U` が格納されます (よくわからない場合は 5.5.6 の `spu_cmpeq` 参照)．そして，その次の "`brz`" が分岐命令です (間に別の命令が挟まっているが無関係なので省略)．細かいことまでは説明しませんが，SPE のアセンブリ言語の表記では，"`br`" で始まる命令が分岐命令になります．`brz` は，指定したレジスタの word0 が 0 だった場合に指定されたアドレスに分岐する分岐命令になります (`brz` の "`z`" は "zero" の "z")．この `brz` はレジスタ$20 を見ていることから，ここでは先ほどの比較の結果が不一致だった場合に分岐が成立するということになります．`k` が 156 でないときはすべて不一致なので，生成された実行コードを走らせると，155/156 の確率で分岐が発生することになります．よって，この `if` 文では，ほぼ毎ループ分岐によるペナルティが生じていることになります．

このような事態を防ぐために，GCC には "`__builtin_expect`" という組み込み関数が用意されています．`__builtin_expect` を `if` 文の条件文の部分に記述することで，その条件文が真になりやすいのか偽になりやすいのかをコンパイラに伝えることができます．

`__builtin_expect` を用いて，この分岐にヒントを与えて分岐が滅多に成立しないことをコンパイラに伝えるには，リスト 6.16 のように `if` の中の条件文を `__builtin_expect` で括ります．

リスト 6.16　滅多に起こらない分岐に分岐ヒントを与えた例

```
    for (i = 0, k = 0; i < num_rand / 4; i++, k++) {
        if (__builtin_expect((k == 156), 0))
            k = 0;
```

`__builtin_expect` は，第 1 引数に条件式を取り，第 2 引数に条件式の結果が何になりがちかを取ります．ここでは条件が偽になりがちということを伝えたいので，第 2 引数に 0 を渡しています．なお，第 2 引数には定数だけでなく動的な変数を指定することも可能です．

`__builtin_expect` 自体は `spu-gcc` 特有のものではなく，他のアーキテクチャ用の GCC でも利用可能で，アーキテクチャに合わせた分岐の最適化を行うようになっています[*11]．`spu-gcc` は，`__builtin_expect` があると，実行時になるべく分岐が成立しなくなるように命令の並びを入れ替えたりする他，有効だと判断した場合は，2.4.2 で紹介した分岐ヒント命令を挿入することも行います．

[*11] Linux のカーネルソースでもあちこちで用いられています．

if 文に分岐ヒントを与えたことで，リスト 6.16 に示すプログラムの実行時間がどの程度変化するか確認してみましょう．分岐ヒントを与えなかった場合，分岐ヒントを与えた場合および誤った分岐ヒントを与えた場合 (“__builtin_expect((k == 156), 1)” と書くことで if 文の条件が真になりやすいというヒントを与えた場合) の 3 つの条件での実行時間を表 6.3 に示します．

表 6.3 分岐ヒントによる性能の変化

状態	実行時間
分岐ヒントなし	52,923,218
分岐ヒントあり	43,009,126
誤った分岐ヒントあり	56,245,758

分岐ヒントを用いて “この if 文の条件式は偽になりがちである” ということをコンパイラに伝えたところ，性能が目に見えて向上したことがわかります．一方，嘘の分岐ヒントを与えた場合，分岐ヒントを与えなかった場合よりも性能が若干悪化しています．分岐ヒントを何も指定しなかった場合でも，悪い方に全力で最適化してしまうというわけでもないようです．

実際に分岐ヒントを与えたことで，生成される実行コードがどのように変化したかを見てみましょう．if 文に対応する部分の逆アセンブル結果をリスト 6.17 に示します．

リスト 6.17 分岐ヒントを与えた部分を逆アセンブルした結果

```
 4d4:   7c 27 02 13     ceqi    $19,$4,156       # 9c
 4d8:   21 00 0c 13     brnz    $19,538 <genrand_mine+0x170>    # 538
```

比較命令は分岐ヒントを与える前とまったく同じですが，その後の分岐命令が “brnz” になりました．brnz は，brz とほぼ同様の分岐命令ですが，指定したレジスタの word0 が 0 でなかった場合に分岐します．よって，このコードで分岐するのはレジスタ$20 に 0 以外の値が入っていたとき，すなわち k が 156 と一致した場合に限られることになり，見事に 1/156 の確率で分岐するコードに置き換わってくれたことがわかります．

3 項演算子への分岐ヒントの適用

ところで，このループの中にはまだ他にも分岐があります．リスト 6.18 に示す部分です．

リスト 6.18 3 項演算子によって分岐が発生する部分

```
        v1b = k < 155 ? vmt[k + 1] : vmt[0];
        v2a = k < 57 ? vmt[k + 99] : vmt[k - 57];
        v2b = k < 56 ? vmt[k + 100] : vmt[k - 56];
```

この部分は if 文ではありませんが，3 項演算子 (?:) が使われているため，分岐命令が生成されると考えられます．今回も実行コードを逆アセンブルして見てみましょう．3 項演算子の処理を行っている部分の逆アセンブル結果をリスト 6.19 に示します．

“cgti” は，レジスタの各ワードスロットと即値の大小関係を調べる命令で，spu_cmpgt に対応する命令の 1 つです．最適化の影響で比較命令とそれに対応する分岐命令が離れていますが，上から順番に k が 56 より大きかったら分岐 (分岐しなかった場合は 57 未満)，そうでなくて (k

リスト 6.19 3 項演算子の部分の実行コード

```
438:   4c 0e 02 17     cgti    $23,$4,56       # 38
...
440:   21 00 18 17     brnz    $23,500 <genrand_mine+0x138>    # 500
...
448:   7c 0e 02 18     ceqi    $24,$4,56       # 38
...
454:   21 00 17 18     brnz    $24,50c <genrand_mine+0x144>    # 50c
...
4e0:   4c 26 82 14     cgti    $20,$4,154      # 9a
...
4f4:   20 7f e6 94     brz     $20,428 <genrand_mine+0x60>     # 428
```

が 56 以下) k が 56 と等しかったら分岐 (分岐しなかった場合は 56 未満)，k が 154 より大きくなかったら分岐という処理になっており，どれもしっかり分岐命令が生成されていることがわかります[*12]．特に 3 つ目の分岐は，k が 155 でない場合に必ず分岐してしまうため，ここでも 155/156 の確率で分岐ペナルティが発生するコードとなっていることがわかります．

そこで，ここにも分岐ヒントを与えてみましょう．3 項演算子にはリスト 6.20 に示すような形で分岐ヒントを与えることができます．この処理では，k が 0 から 155 までの値を等しい確率で取ることが明らかなので，それを元に条件が成立しやすいかどうかを決めて指定しています．

リスト 6.20 3 項演算子に分岐ヒントを与えた例

```
        v1b = __builtin_expect((k < 155), 1) ? vmt[k + 1] : vmt[0];
        v2a = __builtin_expect((k < 57), 0) ? vmt[k + 99] : vmt[k - 57];
        v2b = __builtin_expect((k < 56), 0) ? vmt[k + 100] : vmt[k - 56];
```

実行時間を測定してみましょう．結果を表 6.4 に示します．

表 6.4 3 項演算子への分岐ヒントによる性能の変化

状態	実行時間
3 項演算子への分岐ヒントなし	43,009,126
3 項演算子への分岐ヒントあり	33,717,205

大きく性能が改善していることがわかります．コンパイラによって生成される実行コードを見てみましょう．逆アセンブル結果から関係する部分を抜粋したものをリスト 6.21 に示します．

最適化の影響で比較と分岐が 4 個ずつになってややこしくなっていますが，ここでの処理は以下の通りです．

- k が 154 より大きかったら分岐
- k が 56 より小さかったら分岐
- k が 56 より大きかったら分岐
- k が 56 と等しかったら分岐

よく見ると，3 つ目の分岐は，分岐する確率の方が高くなっています．一見，分岐ヒントが正しく機能していないように見えますが．実行コードをよく見ると，その少し手前にリスト 6.22 に示す形で分岐ヒント命令が挿入されているのがわかります．

[*12] 比較条件が元のコードの表現から若干変化しているのは，SPE には “等しいか” と “大なりか” の 2 種類の比較方法しか用意されていないためです．

リスト 6.21 3 項演算子に分岐ヒントを与えた場合の実行コード

```
438:    4c 26 82 14     cgti    $20,$4,154       # 9a
...
448:    21 00 22 14     brnz    $20,558 <genrand_mine+0x190>    # 558
...
450:    4c 0e 02 17     cgti    $23,$4,56        # 38
...
460:    20 00 1a 17     brz     $23,530 <genrand_mine+0x168>    # 530
...
520:    4c 0e 02 17     cgti    $23,$4,56        # 38
...
52c:    21 7f e7 17     brnz    $23,464 <genrand_mine+0x9c>     # 464
...
538:    7c 0e 02 18     ceqi    $24,$4,56        # 38
...
544:    21 7f e5 98     brnz    $24,470 <genrand_mine+0xa8>     # 470
```

リスト 6.22 分岐ヒント命令

```
504:    12 7f ec 0a     hbrr    52c <genrand_mine+0x164>,464 <genrand_mine+0x9c>
        # 464
...
520:    4c 0e 02 17     cgti    $23,$4,56        # 38
...
52c:    21 7f e7 17     brnz    $23,464 <genrand_mine+0x9c>     # 464
```

“hbrr” というのが分岐ヒント命令になります (SPE のアセンブリ言語の命令表記では，分岐ヒント命令は “hbr” で始まる)．この分岐ヒント命令では，“アドレス 0x52c のところの分岐命令でアドレス 0x464 に飛ぶから準備をしておけ” という指示を出しています．よって，ここでの分岐命令は，他と違って分岐した方がペナルティが少なくなるのです．

このように，if 文や 3 項演算子に対して分岐ヒントを適用することで，分岐の条件が変わったり分岐ヒント命令が挿入されたりして，より分岐ペナルティを減らそうという方向に最適化が進みます．もちろん，分岐ヒントは入れれば入れただけ性能がよくなるというわけでもないので，いつでも今回のように効果を発揮するとは限りません．確実に条件の成否に偏りがあるような場所でも，分岐ヒントを与えたことでかえって性能が悪化してしまうということもあります．必ず入れるべきものと思わずに，入れた場合と入れなかった場合とで比べてみるのがよいでしょう．

ところで，わざわざこのように__builtin_expect を条件式に適用したりせずとも，組み込み関数を利用して分岐ヒント命令を呼び出せばより確実な分岐の最適化ができるのではないかと思われた方がいるかもしれませんが，残念ながらそれはできません．分岐ヒント命令には，対応する組み込み関数が存在しないのです．分岐ヒント命令を入れたかったら，今回のように__builtin_expect を利用した上で，コンパイラに期待するしかありません．なお，分岐ヒント命令も立派な命令の 1 つですから，当然分岐ヒント命令が入ることによって命令数が増加することも忘れないでください．

GCC によるプロファイルフィードバック

spu-gcc43 では，プログラムの実行時にプロファイルを取り，その結果を元に再コンパイルの際にフィードバックをかけて分岐まわりを最適化するプロファイルフィードバックが可能になっています．本書ではコンパイラの機能を活用した分岐の最適化に関しては扱いませんが，興味のある方は調べてみるとよいでしょう．

6.4.2 spu_sel を使った分岐潰し

次に分岐そのものを潰す手法を紹介します．分岐潰しの理屈は簡単です．元々“条件を見てから片方の処理を行う”となっている記述を“とにかく両方演算して，最後に条件に合致した方を残す演算を行う”という記述に書き換えることで分岐を除去します．これを行うのに，第 5 章で紹介した spu_sel を使います．5.5.7 で紹介したように，spu_sel は，3 つのベクタ (2 つの変数と 1 つのパターン) を引数に取って，パターンを元に両方のビットを混ぜたベクタを返す型総称組み込み関数です．このとき，パターンのビットが全部 1 だったり，あるいは全部 0 だったりすると，2 つのベクタ変数をビットごとに混ぜて作ったベクタではなく，元の 2 つのベクタのどちらかの値がそのまま返ることになります．これを利用して分岐を潰します．

たとえばリスト 6.18 の 3 項演算子をリスト 6.23 のように記述することで，分岐そのものをプログラムから排除することができます．

リスト 6.23　分岐を spu_sel に置き換えた状態

```
for (i = 0, k = 0; i < num_rand / 4; i++, k++) {
    vec_int4 vk, vk1, vk2, vk3;
    vec_int4 vp1, vp99, vp100, vm56, vm57;

    if (__builtin_expect((k == 156), 0))
        k = 0;

    v1a = vmt[k];

    vk = spu_promote(k, 0);       // vkのword0にkをロード
    vp1 = spu_add(vk, 1);         // vp1のword0をk+1にする
    vp99 = spu_add(vk, 99);       // vp99のword0をk+99にする
    vp100 = spu_add(vk, 100);     // vp100のword0をk+100にする
    vm56 = spu_add(vk, (int)-56); // vm56のword0をk-56にする
    vm57 = spu_add(vk, (int)-57); // vm56のword0をk-57にする

    // vk1のword0は，vkのword0が154より大きかったら0になり，
    // そうでなければvp1のword0になる
    vk1 = spu_sel(vp1, spu_splats(0), spu_cmpgt(vk, 154));
    vk2 = spu_sel(vp99, vm57, spu_cmpgt(vk, 56));
    vk3 = spu_sel(vp100, vm56, spu_cmpgt(vk, 55));

    // vk1のword0をスカラの扱いにしてvmtのインデックスとして利用する
    v1b = vmt[spu_extract(vk1, 0)];
    v2a = vmt[spu_extract(vk2, 0)];
    v2b = vmt[spu_extract(vk3, 0)];
    ...
```

あまりの豹変ぶりに驚いたかもしれません．順番に見ていきましょう．最初に vk に spu_promote で k を格納しています．spu_promote はスカラをベクタに格上げする型総称組み込み関数でした．ここでは vk の word0 に k の値がそのまま格納され，vk のその他のスロットの内容は不確定となります．

次に，spu_add を用いて vk に数を足したり vk から数を引いたりしています．ここで，元々

あった分岐の条件が成立した場合としなかった場合の両方における vmt のインデックス値を計算しています．

続いて vk1，vk2 および vk3 に spu_sel の戻り値を格納しています．vk1 の行の spu_sel の各引数について確認してみましょう．第 1 引数は vk の各要素に 1 を足したベクタとなります．また，第 2 引数は 0 を引数とした spu_splats の戻り値となっているので，すべての要素が 0 のベクタとなります．そして第 3 引数には vk の各要素と 154 を大小比較した結果が渡されています．vk の word1 から word3 は値が何だかわからないため，この比較の結果の word1 から word3 が 0x00000000 と 0xffffffff のどちらになるのかはわかりませんが，word0 に関しては，vk の word0 が 154 よりも大きかった場合に 0xffffffff となり，そうでなかった場合に 0x00000000 となります．この結果を spu_sel に渡していることから，vk1 の word0 には，vk の word0 が 154 より大きい (“以上” ではない) 場合には 0 が格納され，そうでない場合には vk の word0 に 1 を足した値が格納されることになります．vk の word0 が k だったことを思い返すと，ここまでの処理で，vk1 の word0 には k が 154 より大きい場合に 0 が入り，そうでない場合には k に 1 を足した数が入ることになります．vk2 および vk3 についても同様に考えることができます．

そして最後に vk1 などの word0 を取り出して vmt のインデックスとして用い，得られた値を v1b に代入しています．ここでは，2 つの配列の要素を両方読み込んだ上で，どちらの値を使うのかを選ぶのではなく，配列のインデックスを 2 種類計算しておいて，どちらのインデックスで配列を参照するのかを選択することをしています．これにより，LS からレジスタへのデータのロードのコストを減らしています．

このようにすることで，spu_cmpgt を用いた比較演算はあるものの，分岐を生成する可能性がある 3 項演算子自体をなくすことができます．これが分岐潰しになります．

さて，分岐を潰したことにより，分岐ヒントを入れた場合に比べて性能がどう変化するかを確認してみましょう．実行時間を表 6.5 に示します．

表 6.5　3 項演算子の分岐潰しによる性能の変化

状態	実行時間
3 項演算子への分岐ヒント	33,717,205
3 項演算子の分岐潰し	24,436,860

分岐ヒントを適用した場合よりもさらに処理時間が減りました．手間がかかりましたが，ここでは分岐ヒントを入れるよりも，分岐を潰した方がより効率的であることがわかりました．

ここで，念のため，v1b，v2a および v2b に値をロードする部分の逆アセンブル結果を確認しておきましょう．逆アセンブルしたものをリスト 6.24 に示します．ここでは k にレジスタ$58 が割り当てられています．最初の “il” はレジスタに即値で値を代入する命令です．ここでは 0 をロードしているので，レジスタ$15 はゼロクリアされた状態となっています．次の “ori” はレジスタの各要素と即値の OR を取る命令ですが，ここでは即値が 0 となっているので，単なるレジスタ間の値のコピーとして機能し，レジスタ$22 もゼロクリアされた状態となります．

“ai” はレジスタに即値を足す命令で，“selb” は spu_sel で生成される命令になります．

リスト 6.24 分岐潰しをした部分の実行コード

```
3e8:   40 80 00 0f    il     $15,0
...
41c:   04 00 07 96    ori    $22,$15,0
...
448:   1c 00 5d 3c    ai     $60,$58,1
44c:   4c 26 9d 3d    cgti   $61,$58,154      # 9a
450:   4c 0d dd 38    cgti   $56,$58,55       # 37
454:   87 65 9e 3d    selb   $59,$60,$22,$61
458:   1c 18 dd 33    ai     $51,$58,99       # 63
45c:   0f 61 1d b9    shli   $57,$59,4
460:   1c 19 1d 36    ai     $54,$58,100      # 64
464:   1c f2 1d 37    ai     $55,$58,-56
468:   1c f1 dd 35    ai     $53,$58,-57
...
470:   4c 0e 1d 34    cgti   $52,$58,56       # 38
474:   38 83 9c b0    lqx    $48,$57,$14
478:   86 4d db 38    selb   $50,$54,$55,$56
47c:   86 2d 59 b4    selb   $49,$51,$53,$52
480:   0f 61 19 2f    shli   $47,$50,4
484:   0f 61 18 ae    shli   $46,$49,4
...
498:   38 83 97 aa    lqx    $42,$47,$14
49c:   38 83 97 29    lqx    $41,$46,$14
```

selb の第 2 オペランドから第 4 オペランドは，spu_sel の第 1 引数から第 3 引数に対応します．0x454 の命令を見てみると，0x44c の cgti の結果をパターンとして，0x448 でレジスタ $58 に 1 を足した結果と，ゼロクリアされているレジスタ$22 のどちらかを選ぶようになっています．そして，その結果を 0x45c で 4bit 左シフトすることで，インデックスからアドレスのオフセットに変換しています (“shli” は即値でレジスタを左シフトする命令)．最後に 0x474 の “lqx” 命令で，このアドレスのオフセットを配列の先頭アドレスに足したところから値をロードしています．他の 2 つのベクタ変数に関しても同様に処理されており，実際に分岐を伴わない実行コードが生成されていることがわかります．

コンパイラに自然に分岐を潰させる

ところで，実は今回のソースコードは，先ほどのように必死に spu_sel を埋め込まなくても，リスト 6.25 のように少しまわりに手を加えるだけで，もっと簡単に分岐命令を減らすことができます．

リスト 6.25 簡単に分岐命令を減らす例

```
int k, k1, k2, k3;
...
k = 0;
k1 = 1;   // k+1を常に指す(末尾に到達したら先頭に戻る)
k2 = 99;  // k+99を常に指す(末尾に到達したら先頭に戻る)
k3 = 100; // k+100を常に指す(末尾に到達したら先頭に戻る)
for (i = 0; i < num_rand / 4; i++) { // forの中からkを外に出す
    v1a = vmt[k];
    v1b = vmt[k1];
    v2a = vmt[k2];
    v2b = vmt[k3];

    v1 = spu_shuffle(v1a, v1b, vshuf);
    ...
    vsum = spu_add(vsum, vy2);

    k++;
    if (k == 156)
        k = 0;
    k1++;
```

```
        if (k1 == 156)
            k1 = 0;
        k2++;
        if (k2 == 156)
            k2 = 0;
        k3++;
        if (k3 == 156)
            k3 = 0;
    }
```

これだけです．元々v1b，v2a および v2b の値は，k を演算することで求めたインデックスを用いて vmt を参照することで得ていました．これをやめにして，新たに k1，k2 および k3 の 3 つのインデックス用のカウンタを導入し，個別に 1 周したかどうかを判定するようにしたのがリスト 6.25 になります．リスト 6.25 では分岐ヒントも spu_sel も使っていません．果たしてどれくらい処理性能が変わるのでしょうか．実行時間を測定した結果を表 6.6 に示します．

表 6.6　シンプルな分岐の最適化の性能の変化

状態	実行時間
3 項演算子の分岐潰し	24,436,860
分岐をシンプル化	19,991,253

何と分岐潰しをした場合よりもさらに速くなっています．実際に生成された実行コードを見てみると，ループの中から分岐が消え，代わりにリスト 6.26 に示す命令が生成されていました．

リスト 6.26　シンプルに分岐を減らした場合の条件分岐の部分の実行コード

```
430:    40 80 00 10     il      $16,0
...
434:    0f 61 07 29     shli    $41,$14,4
438:    0f 61 06 bd     shli    $61,$13,4
43c:    0f 61 06 3c     shli    $60,$12,4
440:    0f 61 05 bb     shli    $59,$11,4
...
448:    1c 00 47 0e     ai      $14,$14,1
44c:    38 8a 48 b4     lqx     $52,$17,$41
450:    1c 00 46 8d     ai      $13,$13,1
454:    38 84 5e ba     lqx     $58,$61,$17
458:    1c 00 46 0c     ai      $12,$12,1
45c:    38 84 5e 37     lqx     $55,$60,$17
460:    1c 00 45 a5     ai      $37,$11,1
464:    38 84 5d b8     lqx     $56,$59,$17
468:    7c 27 07 39     ceqi    $57,$14,156      # 9c
46c:    7c 27 06 b6     ceqi    $54,$13,156      # 9c
470:    7c 27 06 33     ceqi    $51,$12,156      # 9c
...
480:    7c 27 12 b1     ceqi    $49,$37,156      # 9c
...
488:    7c 00 1c aa     ceqi    $42,$57,0
...
490:    7c 00 1b 28     ceqi    $40,$54,0
...
4a4:    7c 00 18 a6     ceqi    $38,$49,0
...
498:    7c 00 19 a7     ceqi    $39,$51,0
...
4bc:    81 c3 88 2a     selb    $14,$16,$14,$42
...
4c8:    81 a3 48 28     selb    $13,$16,$13,$40
4cc:    81 83 08 27     selb    $12,$16,$12,$39
4d0:    81 69 48 26     selb    $11,$16,$37,$38
```

よく見ると，登場する命令がリスト 6.24 とほとんど一緒です．ここでは分岐潰しをした場合と同様の実行コードが生成されています．

実は，spu-gcc は，リスト 6.25 のような非常にシンプルな分岐に関しては，このように勝手に selb 命令を使って分岐潰しをしてくれるようになっています．どの程度シンプルであればよいのかというのは筆者もよくわかっていないのですが，条件分岐した後の代入の右辺で演算を行っていない場合 (リテラルや変数名が書かれている場合) に selb になりやすいように思います．

また，余談になりますが，ここでは k が 156 に達していないかどうかの判定をループの最後に持ってきているのもかなり効果を発揮しています．詳しくは次節のソフトウェアパイプライニングの説明で紹介しますが，ループの最初に k を更新する (かもしれない) 処理があると，k の値が決まるまでデータのロードを始めることができなくなるため，命令の並べ替えなどに制約が増えてしまい，最適化がされにくくなります．この問題は，今回のように判定をループの最後に移すことで緩和できます．

なお，リスト 6.26 ではなぜか ceqi が 2 回呼ばれる実行コードが生成されていますが，その理由はよくわかりませんでした．selb の第 2 オペランドと第 3 オペランドのレジスタを入れ替えれば，わざわざそんなことをしなくても済むように思えるのですが ….

6.5 ループの最適化

SPE は，キャッシュがなかったり汎用レジスタが大量にあったりといった特徴を持つことから，性能を引き出すには，一般的なプロセッサと比べてループの最適化にも気を遣う必要があります．以下でループに関連したチューニングのテクニックをいくつか紹介します．

6.5.1 ループの二分

元々のループがどのような数のデータの入力にも対応可能な構造となっている場合，事前にループを二分しておくと最適化しやすくなる場合があります．どのように二分するかというと，キリのよい個数の大部分のデータを処理するループ (メインループ) と，余った少し (端数) を処理するループ (端数処理) の 2 つに分けます．メインループでは，処理対象のうち，キリのよいデータサイズ (256 や 512 など) や処理内容固有の規則性に基づいたデータサイズで割り切れる分を処理します．元のループが中途半端なところから始まって中途半端なところで終わるような処理データにも対応している場合は，ループを三分して，2 番目のループでキリのよい範囲で大部分のデータを処理するようにします．

ここでも，前節までで最適化を適用してきた genrand_mine を題材にして，具体的なループの整理を考えてみます．前節までの genrand_mine は，num_rand を 4 で割った回数ループする単一のループで構成されていました．よって，このループは任意の 4 の倍数個の乱数の処理に対応していることになります．その一方で，元のプログラムの処理を思い返すと，元のプログラムでは 624 回に 1 回の頻度で 624 要素の mt を更新し，また 624 個ある mt の各要素から毎ループ 1 個の乱数を生成していました．よって，元のプログラムでは 624 回のループで全要素を使い

切っていたことになり，このことから，このプログラムには元々624 回ごとに同じ処理が繰り返される周期性があったことがわかります．SIMD 化の過程でループを整理した際に，この周期性を見えにくい形にしてしまいましたが，ループの中で配列のインデックスが 156 に到達したら 0 に戻すという処理を行うのは 624 個データを処理する間の 1 回だけなので，前節の最後の状態でもまだ周期性が残っていると言えます．

この周期性を利用して，ループをメインループと端数処理の 2 つに分けてみましょう．メインループでは，周期性を利用して，num_rand 個の乱数のうちの最初の 624 で割り切れる数を処理します．そして端数処理では num_rand を 624 で割った余りの分の乱数を処理します．ループが 2 つに増えてプログラムが縦に長くなってしまうのですが，従来の “いつどこで終わるのかわからないループ” と違い，メインループは必ず 624 の倍数で終わることが確実となるため，ループの中の処理を最適化しやすくなります．実行時間の大部分を占めるのは大きなメインループの方なので，ここを最適化しやすい形にすることで，全体的な実行時間を短縮しやすいようにできるというわけです．

リスト 6.25 をベースにループを 2 つに分割したものをリスト 6.27 に示します．

リスト 6.27　2 つに分けたループ

```
int i;
int k, k1, k2, k3;
const vec_uchar16 vshuf = (vec_uchar16){4, 5, 6, 7, 8, 9, 10, 11,
                                        12, 13, 14, 15, 16, 17, 18, 19};
const vec_uint4 vone = spu_splats(0x1U);
const vec_uint4 vsel = spu_splats(0x7fffffffU);
vec_uint4 *vmt = (vec_uint4 *)mt;
vec_uint4 v1a, v1b, v2a, v2b, v1, v2;
vec_uint4 vmag01;
vec_uint4 vsum = spu_splats(0U);
vec_uint4 vy, vy2;
int loop1 = num_rand / 624; // メインループのループ回数
int loop2 = num_rand - loop1 * 624; // 端数処理のループ回数

// メインループは1回のループで624要素を処理する
for (i = 0; i < loop1; i++) {
    k1 = 1;
    k2 = 99;
    k3 = 100;
    // 内側のループでは1回のループで4要素をまとめて処理
    for (k  = 0; k < 156; k++) {
        v1a = vmt[k];
        v1b = vmt[k1];
        v2a = vmt[k2];
        v2b = vmt[k3];

        v1 = spu_shuffle(v1a, v1b, vshuf);
        v2 = spu_shuffle(v2a, v2b, vshuf);
        vy = spu_sel(v1a, v1, vsel);
        vmag01 = spu_and(spu_cmpeq(spu_and(vy, vone), vone), 0x9908b0dfU);
        vmt[k] = spu_xor(spu_xor(v2, spu_rlmask(vy, -1)), vmag01);

        vy2 = vmt[k];
        vy2 = spu_xor(vy2, spu_rlmask(vy2, -11));
        vy2 = spu_xor(vy2, spu_and(spu_sl(vy2, 7), 0x9d2c5680U));
        vy2 = spu_xor(vy2, spu_and(spu_sl(vy2, 15), 0xefc60000U));
        vy2 = spu_xor(vy2, spu_rlmask(vy2, -18));
        vsum = spu_add(vsum, vy2);

        k1++;
        if (k1 == 156)
            k1 = 0;
        k2++;
        if (k2 == 156)
            k2 = 0;
```

```
            k3++;
            if (k3 == 156)
                k3 = 0;
        }
    }

    // 端数処理
    k1 = 1;
    k2 = 99;
    k3 = 100;

    // 端数処理はメインループの内側のループと同じ
    for (k = 0; k < loop2 / 4; k++) {
        ....
    }
    vsum = spu_add(vsum, spu_rlqwbyte(vsum, 8));
    vsum = spu_add(vsum, spu_rlqwbyte(vsum, 4));
    return spu_extract(vsum, 0);
```

終了条件で `loop1` を見ている `for` ループがメインループになります．メインループは，624個の乱数を処理するループを `loop1` 回繰り返す二重のループとなっています．一方，終了条件で `loop2` を見ている `for` ループは端数処理となります．端数処理のループの中が省略されていますが，ここにはメインループの内側のループの中身とまったく同じ記述が入ります．

このようなことをしても記述が複雑になるだけで，かえってコンパイラの生成する実行コードの質は落ちるのではないかと思われるかもしれませんが，実際にこのループを組み替えただけのプログラムの実行時間を見てみると，ほとんど影響がないことがわかります．

表6.7 ループを2つに分けた際の性能の変化

状態	実行時間
ループ2分割前	19,991,253
ループ2分割後	19,734,957

ここからは，リスト6.27のメインループを例に，ループの最適化手法を紹介していきます．

SIMD化の際のループ分割

本来であればループを二分 (もしくは三分) する作業はループ内の処理をSIMD化する際にも考える必要があります．例として扱っているHack the Cell 2009の課題では，処理すべきデータの個数が4の倍数ということが確定しており，なおかつ `mt` が16Byte境界にアラインされたアドレスから配置されているため，SIMD化しても端数処理が生じないことから，本書では飛ばしてしまいました．

6.5.2 ループアンローリング

ループアンローリングは，1回のループで行う処理を増やすことでループの回数を減らしたり，あるいはループ回数があらかじめ決まっていて回数が少ない場合にループ自体を除去したりする最適化手法です．一般的に，ループアンローリングを行ってループの回数を減らしたりループそのものを除去したりすることで，条件分岐などの頻度を減らしたり，依存のない範囲でデータの処理の順番を並べ替えたりできるようになるため，処理効率が上がる可能性があります．もちろん，ループを展開することでプログラムの実行局所性が失われてしまうため，ループが大きくな

りすぎた場合は命令キャッシュのミス率が上がるなどして性能が劣化してしまう場合もあります．また，ループを除去したり，ループ 1 回の処理を長くしたりすることで，実行コードが大きくなってしまうという問題もあります．

しかしながら，SPE に関しては，そもそもキャッシュがないため，ループアンローリングをして実行局所性が下がったところで性能が低下する心配はありません．また，ループ内で 1 回に行う処理を増やすことで，ループ内の命令の並べ替えの自由度が増し，コンパイラが最適化を行う際に依存のない命令を並べ替えて詰め込みやすくなるため，前の命令の結果が出るまでパイプラインがストールする (次の命令を発行するまでに待ちが入る) といった無駄を減らすことによる性能向上も期待できます．実行コードのサイズにだけは注意が必要ですが，このような理由などから，余裕があるうちは積極的にループアンローリングすることが推奨されています．

なお，ループアンローリングによる最適化は，GCC も標準でサポートしており，コンパイル時のオプションに`-funroll-loops` をつけるだけで，効果がありそうな所に対して適当にループアンローリングを適用してくれるのですが，ここではそれに頼らずに人間が手と頭を使ってループアンローリングを行います．

ループアンローリングの流れ

いきなり `genrand_mine` のループをアンローリングするのも大変なので，まずはリスト 6.28 の簡単なループを手動でループアンローリングすることを考えてみます．

リスト 6.28 ループアンローリングする前のコード

```
vec_float4 *s1, *s2, *s3, *d;
 ...
for (i = 0; i < n; i++) {
    ...
    d[i] = spu_madd(s1[i], s2[i], s3[i]);
    ...
}
```

リスト 6.28 のループの中では，`s1`，`s2`，`s3` のそれぞれが指す LS 上のデータを読み出し，これに `spu_madd` を適用した結果を `d` の指す領域に書き出す処理を行っています．毎ループ，各配列から 1 要素ずつ読み出して演算し，結果を 1 要素分書き出しています．このようなループに対して，ループの回数を減らして，代わりに 1 回のループで行う処理を増やすのがループアンローリングになります．

ループアンローリングを行う場合は，まず，元のループの何回分を 1 回のループで処理するように展開するのかを考えます．多ければ多いほどよいかというと，そういうわけでもなく，多すぎると逆にレジスタが不足したり，コンパイラがうまく最適化できなくなったりするので，程々のところを狙う必要があります．

次に，ループが元々何回繰り返されるかわからない場合や，繰り返す回数がわかっていてもその回数が展開数で割り切れない場合などは，6.5.1 で述べたようにループをメインループと端数処理に分割します[*13]．

[*13] 元のループが何回繰り返されるかあらかじめわかっている場合は，その約数で展開すれば端数処理が不要になるので，4 や 8 などのキリのよい数にこだわらずに柔軟に展開しましょう．

後はひたすら元のループの処理を並べて書きます．その際，配列のインデックス値をループの中でインクリメントするような記述にすると，ループ中で演算が増えてしまうので，なるべく (ループカウンタ + 定数) の形にするのがよいでしょう (こうするとインデックスを計算する命令が不要になる … かもしれないので).

リスト 6.28 のループについて，元のループ 4 回分の処理を 1 回のループで行うようにアンローリングした場合，リスト 6.29 のようなコードになります．この例では，n が元々どのような数かはっきりしないため，6.5.1 で述べた端数処理も行っています．

リスト 6.29 ループアンローリングの例

```
vec_float4 *s1, *s2, *s3, *d;
...
for (i = 0; i < n - 3; i += 4) {
    ...
    d[i]     = spu_madd(s1[i],     s2[i],     s3[i]);
    d[i + 1] = spu_madd(s1[i + 1], s2[i + 1], s3[i + 1]);
    d[i + 2] = spu_madd(s1[i + 2], s2[i + 2], s3[i + 2]);
    d[i + 3] = spu_madd(s1[i + 3], s2[i + 3], s3[i + 3]);
    ...
}

for (; i < n; i++) {
    ...
    d[i] = spu_madd(s1[i], s2[i], s3[i]);
    ...
}
```

なお，記述がすっきりするからといって，ループアンローリングをした後に，リスト 6.30 のように内側にもう 1 個ループを作り，その中でインデックス値を増やすような書き換えはしないでください．ループアンローリングした意味がなくなるどころか，無駄な処理が増えて，かえってものすごく遅くなってしまいます*14.

リスト 6.30 誤ったループアンローリングの例

```
vec_float4 *s1, *s2, *s3, *d;
...
for (i = 0; i < n - 3; i += 4) {
    int j;
    for (j = 0; j < 4; j++) {
        ...
        d[i + j] = spu_madd(s1[i + j], s2[i + j], s3[i + j]);
        ...
    }
}

for (; i < n; i++) {
    ...
    d[i] = spu_madd(s1[i], s2[i], s3[i]);
    ...
}
```

ループアンローリングの例

次に，これをリスト 6.27 の genrand_mine に適用してみましょう．リスト 6.27 において，前半のメインループの内側のループは 156 回繰り返されることが明らかなので，1 回のループで行う演算量を元のループの 6 回分にすればループの回数は 26 回に減り，13 回分にすれば 12 回に

*14 ただし，そうはならない場合もあります．詳しくは後で具体的なループアンローリングの例を紹介する際にあわせて示します．

減らすことができます．今回は控え目に元のループの 4 回分の処理を 1 回のループで行うようにループアンローリングしてみます[*15]．4 回分のループを 1 回で処理することになるため，これまで SIMD 化によって 1 回のループで `mt` の要素を 4 要素処理していたところが，16 要素を処理するようになります．

ループアンローリングにより，元のループの 4 回分の処理を 1 回のループで行うようにしたメインループをリスト 6.31 に示します．`k1` や `k2` などのインデックスは 4 の倍数ではないことから，何もケアしないとループの途中で 156 以上の値となってしまい，`vmt` の外のデータをアクセスしてしまうことになるため，ループの途中でインデックス値を巻き戻すことをしています．それ以外の部分は，元のループの処理を 4 個分並べ，それぞれで 1 つずつずれたインデックス値で配列を参照するようにすることで，元のループ 4 回分の処理を 1 回のループで行うようにしています．

リスト 6.31　`genrand_mine` のメインループをループアンローリングした例

```
for (i = 0; i < loop1; i++) {
    k1 = 1;
    k2 = 99;
    k3 = 100;

    // ループの回数は1/4にする（終了条件は変えずにkを毎ループ4増やす）
    for (k = 0; k < 156; k += 4) {
        // 元のループk回目相当の処理
        v1a = vmt[k];
        v1b = vmt[k1];
        v2a = vmt[k2];
        v2b = vmt[k3];

        v1 = spu_shuffle(v1a, v1b, vshuf);
        v2 = spu_shuffle(v2a, v2b, vshuf);
        vy = spu_sel(v1a, v1, vsel);
        vmag01 = spu_and(spu_cmpeq(spu_and(vy, vone), vone), 0x9908b0dfU);
        vmt[k] = spu_xor(spu_xor(v2, spu_rlmask(vy, -1)), vmag01);

        vy2 = vmt[k];
        vy2 = spu_xor(vy2, spu_rlmask(vy2, -11));
        vy2 = spu_xor(vy2, spu_and(spu_sl(vy2, 7), 0x9d2c5680U));
        vy2 = spu_xor(vy2, spu_and(spu_sl(vy2, 15), 0xefc60000U));
        vy2 = spu_xor(vy2, spu_rlmask(vy2, -18));
        vsum = spu_add(vsum, vy2);

        // 元のループk+1回目相当の処理
        if (k2 == 155)
            k2 = -1;
        v1a = vmt[k + 1];
        v1b = vmt[k1 + 1];
        v2a = vmt[k2 + 1];
        v2b = vmt[k3 + 1];

        v1 = spu_shuffle(v1a, v1b, vshuf);
        v2 = spu_shuffle(v2a, v2b, vshuf);
        vy = spu_sel(v1a, v1, vsel);
        vmag01 = spu_and(spu_cmpeq(spu_and(vy, vone), vone), 0x9908b0dfU);
        vmt[k + 1] = spu_xor(spu_xor(v2, spu_rlmask(vy, -1)), vmag01);

        vy2 = vmt[k + 1];
        vy2 = spu_xor(vy2, spu_rlmask(vy2, -11));
        vy2 = spu_xor(vy2, spu_and(spu_sl(vy2, 7), 0x9d2c5680U));
        vy2 = spu_xor(vy2, spu_and(spu_sl(vy2, 15), 0xefc60000U));
        vy2 = spu_xor(vy2, spu_rlmask(vy2, -18));
        vsum = spu_add(vsum, vy2);

        // 元のループk+2回目相当の処理
```

[*15] 本当もう少し大きく展開した方が性能が出るのですが，ページ数節約のため 4 回分で我慢します．

```
            v1a = vmt[k + 2];
            v1b = vmt[k1 + 2];
            v2a = vmt[k2 + 2];
            v2b = vmt[k3 + 2];

            v1 = spu_shuffle(v1a, v1b, vshuf);
            v2 = spu_shuffle(v2a, v2b, vshuf);
            vy = spu_sel(v1a, v1, vsel);
            vmag01 = spu_and(spu_cmpeq(spu_and(vy, vone), vone), 0x9908b0dfU);
            vmt[k + 2] = spu_xor(spu_xor(v2, spu_rlmask(vy, -1)), vmag01);

            vy2 = vmt[k + 2];
            vy2 = spu_xor(vy2, spu_rlmask(vy2, -11));
            vy2 = spu_xor(vy2, spu_and(spu_sl(vy2, 7), 0x9d2c5680U));
            vy2 = spu_xor(vy2, spu_and(spu_sl(vy2, 15), 0xefc60000U));
            vy2 = spu_xor(vy2, spu_rlmask(vy2, -18));
            vsum = spu_add(vsum, vy2);

            // 元のループk+3回目相当の処理
            if (k1 == 153)
                k1 = -3;
            v1a = vmt[k + 3];
            v1b = vmt[k1 + 3];
            v2a = vmt[k2 + 3];
            v2b = vmt[k3 + 3];

            v1 = spu_shuffle(v1a, v1b, vshuf);
            v2 = spu_shuffle(v2a, v2b, vshuf);
            vy = spu_sel(v1a, v1, vsel);
            vmag01 = spu_and(spu_cmpeq(spu_and(vy, vone), vone), 0x9908b0dfU);
            vmt[k + 3] = spu_xor(spu_xor(v2, spu_rlmask(vy, -1)), vmag01);

            vy2 = vmt[k + 3];
            vy2 = spu_xor(vy2, spu_rlmask(vy2, -11));
            vy2 = spu_xor(vy2, spu_and(spu_sl(vy2, 7), 0x9d2c5680U));
            vy2 = spu_xor(vy2, spu_and(spu_sl(vy2, 15), 0xefc60000U));
            vy2 = spu_xor(vy2, spu_rlmask(vy2, -18));
            vsum = spu_add(vsum, vy2);

            if (k3 == 152)
                k3 = -4;

            k1 += 4;
            k2 += 4;
            k3 += 4;
        }
    }
```

それではこのコードを実行して，どれくらい速度が改善したかを確認してみましょう．測定した結果を表 6.8 に示します．

表 6.8 ループアンローリングによる性能の変化

状態	実行時間
ループアンローリング前	19,734,957
ループアンローリング後	18,632,878

記述量を 4 倍に増やしたのに，1 割も速くなりませんでした．今回の genrand_mine のメインループのような記述の場合，配列の読み書きの処理については 4 ループ分の間で依存がないため柔軟に命令の並び替えなどが可能になるのですが，途中の演算で用いている変数が共通であるためか，どうもループの中身が大きくなってもコンパイラによる命令の並べ替えや詰め込みなどの最適化が十分に行われないようです．このような場合にループアンローリングの効果を引き出すには，次で紹介するリネーミングを組み合わせます．

6.5.3　変数のリネーミング

spu-gcc は，最適化の際に，ループなどのブロック内で依存関係のない命令間の並び順を調整し，なるべく命令を詰め込めるように努力します．その際，ループなどのブロック内で何度も同じ変数が繰り返し使用されるような記述があると，その変数に関わる命令を自由に動かせなくなってしまい，最適化が進みにくくなる場合があります．これに対し，特に意味なく同じ変数を使用している部分を，処理の結果が変わらない範囲で異なる変数に振り分けると，これらが異なるレジスタに割り振られるようになるため，命令間の依存が減り，コンパイラがより柔軟に最適化を行えるようになります．これをリネーミングと呼びます*[16]．リネーミングはレジスタ間の依存を解消するためにハードウェアで行われるものが有名ですが (もちろん SPE にはそういった機能はない)，ここではそれを C 言語レベルで静的に行おうというわけです．

リネーミングはループアンローリングを行う際にあわせて適用することができます．たとえばリスト 6.31 において，v1a は 1 回のループの中で 4 回同じ処理に使われていますが，毎回最初に値をロードして中身が上書きされるため，これら 4 回の処理で同一の変数を用いる必要はありません．そこで，v1a は，v1a[4] のように 4 要素の配列として宣言し，4 回の各処理で異なるインデックスの要素を利用するようにします．配列にしたらメモリアクセスが頻発してしまうのではないかと心配になるかもしれませんが，spu-gcc はさすがにその辺は心得ていて，あまりにレジスタの使用量が多くてレジスタに常時割り付けられないような場合や volatile をつけて宣言した場合などを除いては，ループ中はこれらの配列の要素はきちんとレジスタに割り付けられるようになっています*[17]．SPE は汎用レジスタを 128 本も持っているので，このようにレジスタを大量に使うような変換を行ってもメモリへのレジスタの退避が発生しにくく，メモリアクセスが頻発してかえって性能が劣化するという事態に陥りにくいという特徴があります．

説明が長くなりましたが，リスト 6.31 のメインループに変数のリネーミングを適用した例をリスト 6.32 に示します．

リスト 6.32　ループアンローリング後に変数をリネーミングした例

```
vec_uint4 *vmt = (vec_uint4 *)mt;
// 変数の使いまわしをやめて，すべて別の変数として宣言する
vec_uint4 v1a[4], v1b[4], v2a[4], v2b[4], v1[4], v2[4];
vec_uint4 vmag01[4];
vec_uint4 vsum = spu_splats(0U);
vec_uint4 vy[4], vy2[4];
...
for (i = 0; i < loop1; i++) {
    k1 = 1;
    k2 = 99;
    k3 = 100;

    for (k = 0; k < 156; k += 4) {
        v1a[0] = vmt[k];
        v1a[1] = vmt[k + 1];
        v1a[2] = vmt[k + 2];
        v1a[3] = vmt[k + 3];
```

*16 本来は手動でこんなことをしなくてもコンパイラが何とかしてくれそうなように思えるのですが，規模が大きい場合などにはうまくいかないことがあるようです．

*17 それでも心配というならば，v1a[4] とせずに，v1a_0，v1a_1，v1a_2，v1a_3 といった風に，きっちり変数の形で宣言してもよいでしょう．

```
v1b[0] = vmt[k1];
v1b[1] = vmt[k1 + 1];
v1b[2] = vmt[k1 + 2];
if (k1 == 153)
    k1 = -3;
v1b[3] = vmt[k1 + 3];

v2a[0] = vmt[k2];
if (k2 == 155)
    k2 = -1;
v2a[1] = vmt[k2 + 1];
v2a[2] = vmt[k2 + 2];
v2a[3] = vmt[k2 + 3];

v2b[0] = vmt[k3];
v2b[1] = vmt[k3 + 1];
v2b[2] = vmt[k3 + 2];
v2b[3] = vmt[k3 + 3];
if (k3 == 152)
    k3 = -4;

v1[0] = spu_shuffle(v1a[0], v1b[0], vshuf);
v1[1] = spu_shuffle(v1a[1], v1b[1], vshuf);
v1[2] = spu_shuffle(v1a[2], v1b[2], vshuf);
v1[3] = spu_shuffle(v1a[3], v1b[3], vshuf);

v2[0] = spu_shuffle(v2a[0], v2b[0], vshuf);
v2[1] = spu_shuffle(v2a[1], v2b[1], vshuf);
v2[2] = spu_shuffle(v2a[2], v2b[2], vshuf);
v2[3] = spu_shuffle(v2a[3], v2b[3], vshuf);

vy[0] = spu_sel(v1a[0], v1[0], vsel);
vy[1] = spu_sel(v1a[1], v1[1], vsel);
vy[2] = spu_sel(v1a[2], v1[2], vsel);
vy[3] = spu_sel(v1a[3], v1[3], vsel);

vmag01[0] = spu_and(spu_cmpeq(spu_and(vy[0], vone), vone),
                    0x9908b0dfU);
vmag01[1] = spu_and(spu_cmpeq(spu_and(vy[1], vone), vone),
                    0x9908b0dfU);
vmag01[2] = spu_and(spu_cmpeq(spu_and(vy[2], vone), vone),
                    0x9908b0dfU);
vmag01[3] = spu_and(spu_cmpeq(spu_and(vy[3], vone), vone),
                    0x9908b0dfU);

vy2[0] = spu_xor(spu_xor(v2[0], spu_rlmask(vy[0], -1)), vmag01[0]);
vy2[1] = spu_xor(spu_xor(v2[1], spu_rlmask(vy[1], -1)), vmag01[1]);
vy2[2] = spu_xor(spu_xor(v2[2], spu_rlmask(vy[2], -1)), vmag01[2]);
vy2[3] = spu_xor(spu_xor(v2[3], spu_rlmask(vy[3], -1)), vmag01[3]);

vmt[k]     = vy2[0];
vmt[k + 1] = vy2[1];
vmt[k + 2] = vy2[2];
vmt[k + 3] = vy2[3];

vy2[0] = spu_xor(vy2[0], spu_rlmask(vy2[0], -11));
vy2[1] = spu_xor(vy2[1], spu_rlmask(vy2[1], -11));
vy2[2] = spu_xor(vy2[2], spu_rlmask(vy2[2], -11));
vy2[3] = spu_xor(vy2[3], spu_rlmask(vy2[3], -11));

vy2[0] = spu_xor(vy2[0], spu_and(spu_sl(vy2[0], 7), 0x9d2c5680U));
vy2[1] = spu_xor(vy2[1], spu_and(spu_sl(vy2[1], 7), 0x9d2c5680U));
vy2[2] = spu_xor(vy2[2], spu_and(spu_sl(vy2[2], 7), 0x9d2c5680U));
vy2[3] = spu_xor(vy2[3], spu_and(spu_sl(vy2[3], 7), 0x9d2c5680U));

vy2[0] = spu_xor(vy2[0], spu_and(spu_sl(vy2[0], 15), 0xefc60000U));
vy2[1] = spu_xor(vy2[1], spu_and(spu_sl(vy2[1], 15), 0xefc60000U));
vy2[2] = spu_xor(vy2[2], spu_and(spu_sl(vy2[2], 15), 0xefc60000U));
vy2[3] = spu_xor(vy2[3], spu_and(spu_sl(vy2[3], 15), 0xefc60000U));

vy2[0] = spu_xor(vy2[0], spu_rlmask(vy2[0], -18));
vy2[1] = spu_xor(vy2[1], spu_rlmask(vy2[1], -18));
vy2[2] = spu_xor(vy2[2], spu_rlmask(vy2[2], -18));
vy2[3] = spu_xor(vy2[3], spu_rlmask(vy2[3], -18));

vsum = spu_add(vsum, vy2[0]);
```

```
            vsum = spu_add(vsum, vy2[1]);
            vsum = spu_add(vsum, vy2[2]);
            vsum = spu_add(vsum, vy2[3]);

            k1 += 4;
            k2 += 4;
            k3 += 4;
        }
    }
```

リスト 6.31 と変数の宣言などを見比べるとわかりますが，とにかくあらゆる変数が 4 要素の配列に置き換えられています．一方，プログラムの記述に関しても，リスト 6.31 とだいぶ雰囲気が変わっているように見えますが，こちらについては，リスト 6.31 では同じ処理をブロック単位で 4 個分繰り返して記述していたところを，リスト 6.32 では行単位で 4 個ずつ記述しているために違って見えるだけで，やっていることはまったく変わりありません．

さて，効果を見てみましょう．処理時間の変化を表 6.9 に示します．

表 6.9　変数のリネーミングによる性能の変化

状態	実行時間
変数のリネーミングなし	18,632,878
変数のリネーミングあり	11,647,613

効果は抜群です．1.5 倍以上高速化されました．かなり手間がかかりましたが，こんな単純な変数のリネーミングでも spu-gcc にとっては格段に最適化しやすくなるようです．

ところで，実はこんなに手間をかけなくても，リスト 6.33 のように記述すると，先ほど頑張ってループアンローリングして変数をリネーミングしたのと同じくらいの効果が得られます．この例は，リスト 6.30 に示した “やってはいけない例” を実践したものになります．これを素直に spu-gcc で-O3 オプションをつけてコンパイルするとものすごく遅くなるのですが，これに-funroll-loops オプションを追加すると，先ほどのリスト 6.32 を上回る速度で動く実行コードが生成されます．動かした際の処理時間を表 6.10 に示します．

リスト 6.33　内側に固定回数のループを入れた記述

```
    for (i = 0; i < loop1; i++) {
        k1 = 1;
        k2 = 99;
        k3 = 100;

        for (k  = 0; k < 156; k += 4) {
            int j;
            for (j = 0;  j < 4; j++) {

                if ((j == 1) & (k2 == 155))
                    k2 = -1;

                if ((j == 3) & (k1 == 153))
                    k1 = -3;

                v1a = vmt[k + j];
                v1b = vmt[k1 + j];
                v2a = vmt[k2 + j];
                v2b = vmt[k3 + j];

                if ((j == 3) & (k3 == 152))
                    k3 = -4;

                v1 = spu_shuffle(v1a, v1b, vshuf);
```

```
                v2 = spu_shuffle(v2a, v2b, vshuf);
                vy = spu_sel(v1a, v1, vsel);
                vmag01 = spu_and(spu_cmpeq(spu_and(vy, vone), vone), 0x9908b0dfU);
                vmt[k + j] = spu_xor(spu_xor(v2, spu_rlmask(vy, -1)), vmag01);

                vy2 = vmt[k + j];
                vy2 = spu_xor(vy2, spu_rlmask(vy2, -11));
                vy2 = spu_xor(vy2, spu_and(spu_sl(vy2, 7), 0x9d2c5680U));
                vy2 = spu_xor(vy2, spu_and(spu_sl(vy2, 15), 0xefc60000U));
                vy2 = spu_xor(vy2, spu_rlmask(vy2, -18));
                vsum = spu_add(vsum, vy2);
            }
            k1 += 4;
            k2 += 4;
            k3 += 4;
        }
    }
```

表 6.10 GCC によるループアンローリングの効果

状態	実行時間
手動でループアンローリング・リネーミングしたもの	11,647,613
`-O3` のみでコンパイルした場合	31,951,016
`-O3 -funroll-loops` でコンパイルした場合	11,316,757

どうもこのように 2 重のループにした場合，内側のループの処理の回数が 4 回に固定されるため，GCC がこれを検出して，自動でループアンローリングおよびリネーミング相当の最適化を施してくれるようです*18．この辺りの GCC のクセなども押さえた上で，ループアンローリングや変数のリネーミングを手動で行うかどうかの判断を行うとよいのかもしれません．

6.5.4 ソフトウェアパイプライニング

ソフトウェアパイプライニングは，ループの中での依存関係を減らす最適化手法です．コンパイラは，通常，ループなどのブロック単位で最適化を行います．その際，ループのブロックの中で，メモリからレジスタへのデータのロード → 演算→ レジスタの値のメモリへのストア の順で処理を行っている場合，データのロードが終わらないと演算ができないことから，コンパイラはデータのロードを必ず演算より前に持ってくることを行います．依存関係があるのだから当たり前だと思うかもしれませんが，これを打破するのがソフトウェアパイプライニングになります．

何をするかと言うと，まずループの回数を 1 回減らします．そして 1 回目のロードをループの手前に出します．さらに，ループのブロックの中は，ループの手前で読み込んだデータを演算するところから始め，次にストアし，最後に次にループの先頭に戻った際に処理するデータのロードを行うように書き換えます．ループの後ろには，ループを抜ける直前に行ったロードに対応する演算とストアを付加します．雰囲気的には図 6.4 のようなイメージでループを再構成します．

ループの中では，レジスタ上の値を用いた演算とストア処理の間に依存関係があるため，当然ながら演算が終わるまでストアはできませんが，演算で用いたレジスタが不要になった時点で次のループで使うデータをそこに読み出すことができるため，ロードの処理を演算と並列に行うことが可能になります．これにより，コンパイラによる命令の詰め込み具合を上げて処理速度を上

*18 GCC 4.1.1 ベースの spu-gcc でコンパイルした場合，ここまで速くはなりませんでした．GCC 4.3 系で強化されている部分のようです．

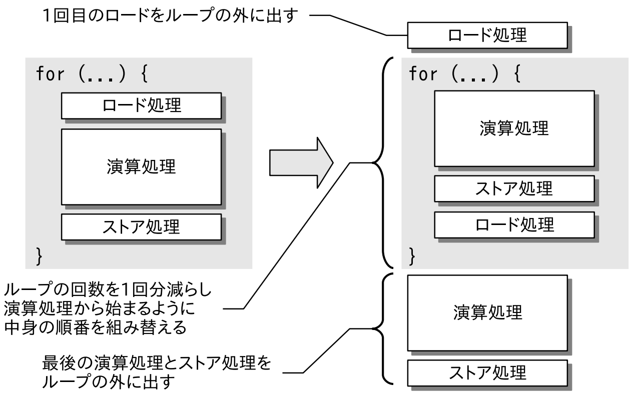

図 6.4 ソフトウェアパイプライニング適用時のイメージ

げようというのがソフトウェアパイプライニングのアイディアになります．

いつもの genrand_mine のメインループにソフトウェアパイプライニングを適用したものをリスト 6.34 に示します．

リスト 6.34 ソフトウェアパイプライニングを適用した例

```
    for (i = 0; i < loop1; i++) {
        k1 = 1;
        k2 = 99;
        k3 = 100;

        // 最初に1回目のループで使うデータをループの外でロードする
        v1a[0] = vmt[0];
        v1a[1] = vmt[1];
        v1a[2] = vmt[2];
        v1a[3] = vmt[3];

        v1b[0] = vmt[1];
        v1b[1] = vmt[2];
        v1b[2] = vmt[3];
        v1b[3] = vmt[4];

        v2a[0] = vmt[99];
        v2a[1] = vmt[100];
        v2a[2] = vmt[101];
        v2a[3] = vmt[102];

        v2b[0] = vmt[100];
        v2b[1] = vmt[101];
        v2b[2] = vmt[102];
        v2b[3] = vmt[103];

        // ループのカウンタの初期値はロードした分ずらしておく
        for (k = 4; k < 156; k += 4) {
            // ループの中は先に演算とストアを持ってくる
            v1[0] = spu_shuffle(v1a[0], v1b[0], vshuf);
            v1[1] = spu_shuffle(v1a[1], v1b[1], vshuf);
            v1[2] = spu_shuffle(v1a[2], v1b[2], vshuf);
            v1[3] = spu_shuffle(v1a[3], v1b[3], vshuf);

            v2[0] = spu_shuffle(v2a[0], v2b[0], vshuf);
            v2[1] = spu_shuffle(v2a[1], v2b[1], vshuf);
            v2[2] = spu_shuffle(v2a[2], v2b[2], vshuf);
            v2[3] = spu_shuffle(v2a[3], v2b[3], vshuf);

            vy[0] = spu_sel(v1a[0], v1[0], vsel);
            vy[1] = spu_sel(v1a[1], v1[1], vsel);
            vy[2] = spu_sel(v1a[2], v1[2], vsel);
            vy[3] = spu_sel(v1a[3], v1[3], vsel);

            vmag01[0] = spu_and(spu_cmpeq(spu_and(vy[0], vone), vone),
                                0x9908b0dfU);
```

```
        vmag01[1] = spu_and(spu_cmpeq(spu_and(vy[1], vone), vone),
                            0x9908b0dfU);
        vmag01[2] = spu_and(spu_cmpeq(spu_and(vy[2], vone), vone),
                            0x9908b0dfU);
        vmag01[3] = spu_and(spu_cmpeq(spu_and(vy[3], vone), vone),
                            0x9908b0dfU);

        vy2[0] = spu_xor(spu_xor(v2[0], spu_rlmask(vy[0], -1)), vmag01[0]);
        vy2[1] = spu_xor(spu_xor(v2[1], spu_rlmask(vy[1], -1)), vmag01[1]);
        vy2[2] = spu_xor(spu_xor(v2[2], spu_rlmask(vy[2], -1)), vmag01[2]);
        vy2[3] = spu_xor(spu_xor(v2[3], spu_rlmask(vy[3], -1)), vmag01[3]);

        vmt[k - 4] = vy2[0];
        vmt[k - 3] = vy2[1];
        vmt[k - 2] = vy2[2];
        vmt[k - 1] = vy2[3];

        vy2[0] = spu_xor(vy2[0], spu_rlmask(vy2[0], -11));
        vy2[1] = spu_xor(vy2[1], spu_rlmask(vy2[1], -11));
        vy2[2] = spu_xor(vy2[2], spu_rlmask(vy2[2], -11));
        vy2[3] = spu_xor(vy2[3], spu_rlmask(vy2[3], -11));

        vy2[0] = spu_xor(vy2[0], spu_and(spu_sl(vy2[0], 7), 0x9d2c5680U));
        vy2[1] = spu_xor(vy2[1], spu_and(spu_sl(vy2[1], 7), 0x9d2c5680U));
        vy2[2] = spu_xor(vy2[2], spu_and(spu_sl(vy2[2], 7), 0x9d2c5680U));
        vy2[3] = spu_xor(vy2[3], spu_and(spu_sl(vy2[3], 7), 0x9d2c5680U));

        vy2[0] = spu_xor(vy2[0], spu_and(spu_sl(vy2[0], 15), 0xefc60000U));
        vy2[1] = spu_xor(vy2[1], spu_and(spu_sl(vy2[1], 15), 0xefc60000U));
        vy2[2] = spu_xor(vy2[2], spu_and(spu_sl(vy2[2], 15), 0xefc60000U));
        vy2[3] = spu_xor(vy2[3], spu_and(spu_sl(vy2[3], 15), 0xefc60000U));

        vy2[0] = spu_xor(vy2[0], spu_rlmask(vy2[0], -18));
        vy2[1] = spu_xor(vy2[1], spu_rlmask(vy2[1], -18));
        vy2[2] = spu_xor(vy2[2], spu_rlmask(vy2[2], -18));
        vy2[3] = spu_xor(vy2[3], spu_rlmask(vy2[3], -18));

        vsum = spu_add(vsum, vy2[0]);
        vsum = spu_add(vsum, vy2[1]);
        vsum = spu_add(vsum, vy2[2]);
        vsum = spu_add(vsum, vy2[3]);

        k1 += 4;
        k2 += 4;
        k3 += 4;

        // その後次のループで使うデータのロードを行う

        v1a[0] = vmt[k];
        v1a[1] = vmt[k + 1];
        v1a[2] = vmt[k + 2];
        v1a[3] = vmt[k + 3];

        v1b[0] = vmt[k1];
        v1b[1] = vmt[k1 + 1];
        v1b[2] = vmt[k1 + 2];
        if (k1 == 153)
            k1 = -3;
        v1b[3] = vmt[k1 + 3];

        v2a[0] = vmt[k2];
        if (k2 == 155)
            k2 = -1;
        v2a[1] = vmt[k2 + 1];
        v2a[2] = vmt[k2 + 2];
        v2a[3] = vmt[k2 + 3];

        v2b[0] = vmt[k3];
        v2b[1] = vmt[k3 + 1];
        v2b[2] = vmt[k3 + 2];
        v2b[3] = vmt[k3 + 3];

        if (k3 == 152)
            k3 = -4;
    }
```

```
        // ループの最後にロードしたデータの演算とストアをループの外で行う
        v1[0] = spu_shuffle(v1a[0], v1b[0], vshuf);
        v1[1] = spu_shuffle(v1a[1], v1b[1], vshuf);
        v1[2] = spu_shuffle(v1a[2], v1b[2], vshuf);
        v1[3] = spu_shuffle(v1a[3], v1b[3], vshuf);

        v2[0] = spu_shuffle(v2a[0], v2b[0], vshuf);
        v2[1] = spu_shuffle(v2a[1], v2b[1], vshuf);
        v2[2] = spu_shuffle(v2a[2], v2b[2], vshuf);
        v2[3] = spu_shuffle(v2a[3], v2b[3], vshuf);

        vy[0] = spu_sel(v1a[0], v1[0], vsel);
        vy[1] = spu_sel(v1a[1], v1[1], vsel);
        vy[2] = spu_sel(v1a[2], v1[2], vsel);
        vy[3] = spu_sel(v1a[3], v1[3], vsel);

        vmag01[0] = spu_and(spu_cmpeq(spu_and(vy[0], vone), vone), 0x9908b0dfU);
        vmag01[1] = spu_and(spu_cmpeq(spu_and(vy[1], vone), vone), 0x9908b0dfU);
        vmag01[2] = spu_and(spu_cmpeq(spu_and(vy[2], vone), vone), 0x9908b0dfU);
        vmag01[3] = spu_and(spu_cmpeq(spu_and(vy[3], vone), vone), 0x9908b0dfU);

        vy2[0] = spu_xor(spu_xor(v2[0], spu_rlmask(vy[0], -1)), vmag01[0]);
        vy2[1] = spu_xor(spu_xor(v2[1], spu_rlmask(vy[1], -1)), vmag01[1]);
        vy2[2] = spu_xor(spu_xor(v2[2], spu_rlmask(vy[2], -1)), vmag01[2]);
        vy2[3] = spu_xor(spu_xor(v2[3], spu_rlmask(vy[3], -1)), vmag01[3]);

        vmt[152] = vy2[0];
        vmt[153] = vy2[1];
        vmt[154] = vy2[2];
        vmt[155] = vy2[3];

        vy2[0] = spu_xor(vy2[0], spu_rlmask(vy2[0], -11));
        vy2[1] = spu_xor(vy2[1], spu_rlmask(vy2[1], -11));
        vy2[2] = spu_xor(vy2[2], spu_rlmask(vy2[2], -11));
        vy2[3] = spu_xor(vy2[3], spu_rlmask(vy2[3], -11));

        vy2[0] = spu_xor(vy2[0], spu_and(spu_sl(vy2[0], 7), 0x9d2c5680U));
        vy2[1] = spu_xor(vy2[1], spu_and(spu_sl(vy2[1], 7), 0x9d2c5680U));
        vy2[2] = spu_xor(vy2[2], spu_and(spu_sl(vy2[2], 7), 0x9d2c5680U));
        vy2[3] = spu_xor(vy2[3], spu_and(spu_sl(vy2[3], 7), 0x9d2c5680U));

        vy2[0] = spu_xor(vy2[0], spu_and(spu_sl(vy2[0], 15), 0xefc60000U));
        vy2[1] = spu_xor(vy2[1], spu_and(spu_sl(vy2[1], 15), 0xefc60000U));
        vy2[2] = spu_xor(vy2[2], spu_and(spu_sl(vy2[2], 15), 0xefc60000U));
        vy2[3] = spu_xor(vy2[3], spu_and(spu_sl(vy2[3], 15), 0xefc60000U));

        vy2[0] = spu_xor(vy2[0], spu_rlmask(vy2[0], -18));
        vy2[1] = spu_xor(vy2[1], spu_rlmask(vy2[1], -18));
        vy2[2] = spu_xor(vy2[2], spu_rlmask(vy2[2], -18));
        vy2[3] = spu_xor(vy2[3], spu_rlmask(vy2[3], -18));

        vsum = spu_add(vsum, vy2[0]);
        vsum = spu_add(vsum, vy2[1]);
        vsum = spu_add(vsum, vy2[2]);
        vsum = spu_add(vsum, vy2[3]);
    }
```

さらに倍のサイズに膨れ上がりました．リスト 6.34 では，1 回目のループのロードと最後のループの演算およびストアをループの前後に出しています．ループの外に出た部分は配列のインデックス値などが一意に定まるため，これらを定数としています．ソフトウェアパイプライニングの効果を確認してみましょう．実行時間の測定結果を表 6.11 に示します．

表 6.11　ソフトウェアパイプライニングによる性能の変化

状態	実行時間
ソフトウェアパイプライニングなし	11,647,613
ソフトウェアパイプライニング適用後	10,412,727

1 割以上処理時間が短縮されていることがわかります．処理内容によってはもっと大幅に性能

が改善することもあるでしょう．

6.6 SPU のパイプラインのチューニング

続いてパイプラインのチューニングについて紹介します．5.6 で紹介したように，SPU のパイプラインは，演算ばかりが続くと偶数パイプに命令が偏りがちになるという傾向があります．ここで紹介するパイプラインのチューニングは，まずプログラムの実行コードにおけるこの偏りを把握し，そして何とかして偏りを減らすことで実行時間を短縮しようという手法になります．

6.6.1 パイプラインの使用状況を知る

自分の作成したプログラムがどの程度パイプラインを有効利用できているのかについては，Cell SDK に付属している `spu_timing` というアプリケーションを利用して実行コードを静的に解析することで知ることができます．

`spu_timing` で実行コードの解析を行うには，まず，`spu-gcc` に`-S` オプションをつけてソースコードのコンパイルを行い，アセンブルを行う直前の状態 (C 言語で記述されたプログラムがアセンブリ言語に変換された状態) のコードをファイルに出力します．次に，その出力されたファイルを `spu_timing` コマンドに読ませます．これにより，命令間の依存や命令発行のタイミングなどが静的に解析され，結果が “`.timing`” という拡張子のついたファイル名で出力されます．このとき，ソースコードのコンパイル時に`-g` オプションをつけていると，解析の結果にデバッグ情報が大量に埋め込まれて非常に読みにくくなってしまうので，ここではつけないことをおすすめします．

`spu_timing` で解析を行う際のコマンドの例を以下に示します．ここでは `mt_mine.c` という名前の C 言語のソースコードから生成される実行コードを解析しています．実行すると `mt_mine.s.timing` という名前のファイルが生成されます．

```
% spu-gcc43 -std=gnu99 -O3 -Wall -S mt_mine.c -o mt_mine.s
% /opt/cell/sdk/usr/bin/spu_timing mt_mine.s
```

`.timing` ファイルには，リスト 6.35 のような結果が長々と書き出されています．中身は単なるテキストファイルなので，ターミナル上で `less` などのページャを用いて中身を見ることができるのですが，IBM の開発者は幅が 80 文字のターミナルを使わないためなのか，`spu_timing` の出力は 1 行が長く，80 文字のターミナルだと若干はみ出て折り返して表示されてしまうのが残念なところです．

以下で timing ファイルの読み方について解説します．まず，真ん中の数字は無視して，両端に注目してください．右側にある “`il`” や “`ceq`” などと書かれた部分は，6.2 でも紹介しましたが，C 言語で書いたソースをコンパイルした結果生成された SPE の命令をアセンブリ言語の形式で表記したものになります．“`.L262`” などの `spu-objdump` の出力にはないような記述もありますが，これはラベルです．C 言語の `goto` 文で利用するラベルと同じようなものだと思ってください．どこかにこのラベルの実体があるはずです．また，同じ行の左側の端を見ると，行頭に “0” もしくは “1” という文字がありますが，これがその行の命令がどちらのパイプで実行される

リスト 6.35 spu_timing の出力の例

```
0  0                                                   9           il       $61,64
1  -123456                                                         wrch     $ch21,$61
0     23                                                           il       $60,0
0d     -45                                                         ceq      $59,$118,$60
1d      --6789                                                     gb       $58,$59
0D         ---01                                                   cgti     $57,$58,11
1D            0                                                    hbrp     # 2
1              -2345                                               brnz     $57,.L262
1                3                                                 lnop
0D                45                                               ai       $104,$sp,48
1D                456789                                           lqd      $72,64($sp)
```

かを示しています．0 は偶数パイプ，1 は奇数パイプに対応します．

行によっては，先頭の数字の直後に “D” や “d” と記載されています．先頭が “0D” もしくは “1D” と書かれている行は，実行時に 2 命令同時発行される命令のペアになります．一方，“0d” もしくは “1d” となっている行は，本来 2 命令同時発行できる命令の組み合わせと配置でありながら，使用するレジスタの依存関係などの都合により 2 命令同時に発行することができないペアを表しています．上記の例では，ceq と gb がこれに該当します．このペアの場合，奇数パイプで実行される gb 命令が，偶数パイプで実行される ceq 命令の結果 (レジスタ$59) を入力として使用するため，ceq が終わるまで発行を待つ必要があり，それゆえ 2 命令同時発行ができていません．また，D も d もついていない行は，配置などの問題から，依存があってもなくても 2 命令同時発行できない命令を表しています．

ベストなのは，D のついた命令が延々と続いている状態，すなわち毎サイクル 2 命令同時発行となっている状態です．ただし，D がいっぱいあるように見えて，実は nop や lnop といった何もしない調整用の命令がたくさん含まれている場合もあるので，意味のある演算をやっているのかどうかも，あわせて気にすべきです．

真ん中の空間に現れる数字の並びは，数字自体には特別な意味はありません．数字の個数が，その命令の実行に要するサイクル数を示しています．また，オペランドに依存があるせいで命令の発行が待たされる区間は “-” で表示されます．たとえばリスト 6.35 において，ceq のところには数字が 2 個並んでいるので，ceq は 2 サイクルかかる命令だということがわかります．また，先頭に “-” が 1 個ついていることから，レジスタの依存が解決できずに，発行までに 1 サイクル待たされるということがわかります．その 1 サイクルは，直前の il によってレジスタ$60 の用意が整うまでの待ち時間になります．同時に発行される命令は，数字の並びの開始位置が同一となり，次のサイクルに発行される命令は，1 文字分ずれたところから始まります．数字の並びが右端まで到達してしまうと，続きは左側から現れます．

なお，spu_timing ではなく，Assembly Visualizer という高機能かつグラフィカルなソフトでも同じようなことを確認できるようです．しかしながら，筆者はこのツールを解説できるほど利用したことがないので，本書では扱いません．詳しく知りたい方は Web 上のドキュメントなどを参照してください．

6.6.2 パイプ間の処理のバランスを調整する

続いて，解析の結果を踏まえたチューニングを考えてみます．5.6 ではあまり詳しく述べませんでしたが，SPE の奇数パイプは元々はデータのロードやストアを行うのが主な役割のパイプラインになります．偶数パイプで演算を行いつつ奇数パイプでデータのロード・ストアを行うことで，マルチメディア系のストリーミングデータ処理などで高い性能を発揮することができます．このような状況では，両方のパイプラインに命令がギッチリ詰まっている状態となっており，パイプ間の処理のバランスの調整も何もないのですが，一般的にこのような状態を作ることができるのは演算の量と LS へのアクセスのバランスがうまい具合に取れている場合に限られ，大抵はどちらかに偏ります．また，ロード・ストアと演算処理の量のバランスが取れたプログラムでも，ループの中で依存関係があるせいでムラが生じ，結局両方のパイプが同時に動かせないといったケースも考えられます．これをある程度解消するのが 6.5.4 で紹介したソフトウェアパイプライニングなわけですが，それにも限界があります．

そこで，次に出てくるのが，ここで紹介する，パイプ間で処理を移してバランスを調整しようという話になります．もっとも，パイプ間の処理のバランスを調整すると言っても，ロード・ストアを偶数パイプに移すことは，動的に生成可能なデータでもない限り難しいことから，通常は“演算の量がロード・ストアに比べてやたらと多いせいで偶数パイプに処理が偏っているのを緩和する” というケースが中心となるでしょう．また，その場合でも，5.6 で紹介したように，奇数パイプで実行可能な演算処理は，128bit 幅のレジスタ全体を 1 つのデータとして処理を行うような特殊な系統のものに限られるため，どんな処理でも奇数パイプに移せるというわけではありません．非常に適用できる場合が限られる最適化手法と言えます．

以下で，偶数パイプから奇数パイプに処理を移す具体的な例を紹介します．ここでも，例によって Hack the Cell 2009 の課題のコードを題材とします．前節の最後で紹介したソフトウェアパイプライニング適用後のソースコードをコンパイルすると，メインループの部分はリスト 6.36 のような実行コードとなります (全部は掲載しきれないので一部のみ抜粋)．

この抜き出した部分に限らず，全体的に 2 命令同時発行ができておらず，特に偶数パイプのみの命令発行が続いている点が目につきます．メインループが，元々少ないデータに多くの演算を適用するタイプのものであるためロードやストアが少なく，またメインループの中で用いている演算命令のほとんどが偶数パイプを使っているものであることから，このような状態が発生しているものと考えられます．これを何とかするために，奇数パイプに移すことができる処理がないか考えてみます．

まず簡単に気がつくのはリスト 6.37 に示す `vmag01` を求めるための演算部分でしょう．リスト 6.37 では，`spu_and` で `vy` の LSB のみを取り出し，その上で各要素を `spu_cmpeq` で 1 と比較することで，LSB が 0 だったら対応するスロットに `0x00000000` が入り，LSB が 1 だったら対応するスロットに `0xffffffff` が入るベクタを作りました．

リスト 6.36 genrand_mine のタイミング解析の結果

```
0                 5678                                            rotmi   $4,$11,-11
0                  6789                                           shli    $3,$79,7
0                   78                                            xor     $73,$7,$9
0                    8901                                         shli    $78,$76,7
0                     90                                          xor     $70,$11,$4
0                      01                                         and     $2,$3,$24
0                       1234                                      shli    $75,$73,7
0                        23                                       xor     $67,$79,$2
0                         34                                      and     $77,$78,$24
0                          4567                                   shli    $72,$70,7
0                           56                                    xor     $64,$76,$77
0                            67                                   and     $74,$75,$24
0                             7890                                shli    $69,$67,15
0                              89                                 xor     $61,$73,$74
0                               90                                and     $71,$72,$24
0                                0123                             shli    $66,$64,15
0                                 12                              xor     $58,$70,$71
0                                  23                             and     $68,$69,$31
0                                   3456                          shli    $63,$61,15
0                                    45                           xor     $56,$67,$68
0                                     56                          and     $65,$66,$31
0                                      6789                       shli    $60,$58,15
0                                       78                        xor     $51,$64,$65
0                                        89                       and     $62,$63,$31
0                                         9012                    rotmi   $57,$56,-18
```

リスト 6.37 vmag01 の演算部分

```
vmag01[0] = spu_and(spu_cmpeq(spu_and(vy[0], vone), vone),
                    0x9908b0dfU);
```

実は，この部分の処理は，リスト 6.38 のように奇数パイプだけで実現することが可能です．

リスト 6.38 奇数パイプだけでマスクとなるベクタを生成する処理の例

```
vmag01[0] = spu_and(spu_maskw(spu_extract(spu_gather(vy[0]), 0)),
                    0x9908b0dfU);
```

`spu_gather` で `vy[0]` の各要素の LSB をベクタの word0 スロットに集めた上で，それを `spu_extract` でスカラとし，さらに `spu_maskw` でその下位 4bit の各々を 32bit に広げています．よって，リスト 6.37 と同じビットマスクが生成されることになります．命令数が増えてしまっているように見えるかもしれませんが，この `spu_extract` は，プリファードスロットにある要素を取り出すことがコンパイルの時点で確定しているので，命令に対応する実行コードが生成されることはありません．したがって，このように書き換えることで，偶数パイプの命令 2 個を，奇数パイプの命令 2 個に置き換えられたことになります．

この変更でどれだけ性能が変化するかを確認してみましょう．実行時間の比較を表 6.12 に示します．1 割にも満たない差分ですが，それでも若干性能改善が見られます．

表 6.12 vmag01 の計算に奇数パイプの命令を用いた場合の性能変化

状態	実行時間
偶数パイプ使用時	10,412,727
奇数パイプ使用時	9,776,644

また，偶数パイプを用いたスロット内のシフト・ローテート処理は，元のデータや周辺の演算次第では，奇数パイプを用いたベクタ全体のシフト・ローテート処理に置き換えられる場合があります．リスト 6.39 に示す，Tempering 処理で左シフト結果に各要素のビットマスクを適用している部分も，奇数パイプを使うように書き換えることができます．

リスト 6.39　各要素を左シフトした結果にビットマスクを適用している部分

```
vy2[0] = spu_xor(vy2[0], spu_and(spu_sl(vy2[0], 7), 0x9d2c5680U));
...
vy2[0] = spu_xor(vy2[0], spu_and(spu_sl(vy2[0], 15), 0xefc60000U));
```

この処理では，ベクタの各要素をそれぞれ 7bit および 15bit 左にシフトしています．これをそのままベクタ全体の左シフトで置き換えるには，通常だと右隣の要素から入ってきたビットを除去するために各要素の下位のビットをマスクするなどの処理が必要となりますが，ここでは元々左シフトした各要素に定数を AND するようになっているため，要素単位のシフトをベクタ全体のシフトに置き換えてもマスク処理を追加で行う必要はありません．これを利用して，スロット単位のシフトをベクタ全体のシフトに置き換えたものをリスト 6.40 に示します．

リスト 6.40　スロットごとの左シフトをベクタ全体の左シフトに置き換えた例

```
vy2[0] = spu_xor(vy2[0], spu_and(spu_slqw(vy2[0], 7), 0x9d2c5680U));
...
vy2[0] = spu_xor(vy2[0],
                 spu_and(spu_slqwbytebc(spu_slqw(vy2[0], 15), 15),
                         0xefc60000U));
```

ここでは，ワードスロット内の左シフトを 128bit 全体の左シフトに置き換えました．なお，レジスタをまるごと 8bit 以上左シフトする命令はないため，`qw` 系の命令で 15bit の左シフトを実現するには命令が 2 個必要になります．これだと全体の命令数が増えてしまいますが，元々奇数パイプの方が空いていることから，奇数パイプに新たに 2 命令を追加することになっても全体の性能が悪化することはなく，逆に，これにより処理時間に直結している偶数パイプの命令が 1 つ減るため，それによる全体の処理時間の短縮が期待されます．

実際に性能向上につながるか確認してみましょう．実行時間の測定結果を表 6.13 に示します．

表 6.13　Tempering 処理の一部を奇数パイプで行った際の性能変化

状態	実行時間
偶数パイプ使用時	9,776,644
奇数パイプ使用時	9,066,002

処理時間がさらに短くなっていることがわかります．ここだけで見ると 5% にも満たない差ですが，`vmag01` の導出を奇数パイプラインに移す前と比べると 1 割以上の性能向上となっており，パイプ間の命令の移動もなかなか侮れないことがわかります．

実際に実行コードにどのような変化が生じているのか，`spu_timing` の結果で見てみましょう．命令の一部を奇数パイプに移した際の実行コードのうち，リスト 6.36 に示したのと同じ処理を行っている部分のタイミング解析の結果をリスト 6.41 に示します．リスト 6.36 と比べると，行数が増えているため，かえって遅くなってしまったのではないかと思われそうですが，よく見る

と "D" がついている行 (2 命令同時発行できている部分) が増えています．2 命令同時発行となる部分は 2 行で 1 サイクルとなるため，全体で見るとリスト 6.36 よりも処理時間が短くなっていると言えます (真ん中の数字が何桁分進んだかで所用サイクル数がわかる)．

リスト 6.41 命令の一部を奇数パイプに移した genrand_mine のタイミング解析の結果

```
0D       6789                                                                rotmi    $7,$4,-11
1D       6                                                                   lnop
0D        7890                                                               shli     $11,$13,4
1D        7890                                                               shlqbii  $3,$79,7
0          89                                                                xor      $75,$6,$10
0           90                                                               and      $9,$25,$12
0D           01                                                              xor      $72,$4,$7
1D           0123                                                            shlqbii  $78,$75,7
0D            12                                                             xor      $67,$31,$9
1D            123456                                                         lqx      $31,$11,$20
0D             23                                                            and      $2,$25,$3
1D             2345                                                          shlqbii  $77,$72,7
0D              3456                                                         shli     $71,$14,4
1D              3456                                                         shlqbii  $74,$67,7
0                45                                                          xor      $61,$79,$2
0                 56                                                         and      $76,$25,$78
0D                 67                                                        and      $73,$25,$77
1D                 6789                                                      shlqbii  $70,$61,7
0D                  78                                                       xor      $58,$75,$76
1D                  7890                                                     shlqbyi  $69,$74,1
0D                   89                                                      xor      $53,$72,$73
1D                   890123                                                  lqx      $14,$71,$20
0D                    9012                                                   shli     $64,$17,4
1D                    9012                                                   shlqbii  $66,$58,7
0D                     01                                                    ceqi     $40,$17,153
1D                     0123                                                  shlqbyi  $65,$70,1
0D                      12                                                   and      $68,$33,$69
1D                      1234                                                 shlqbii  $63,$53,7
0D                       23                                                  ai       $18,$18,4
1D                       2                                                   lnop
0D                        34                                                 xor      $55,$67,$68
1D                        3456                                               shlqbyi  $60,$66,1
0D                         45                                                and      $62,$33,$65
1D                         456789                                            lqx      $13,$64,$20
0D                          5678                                             rotmi    $56,$55,-18
```

筆者は散々頭をひねっても，奇数パイプに移して全体の処理の効率化につながりそうな演算をこの 2 つ以外で見つけることができませんでしたが，もしかしたら他にも奇数パイプを使うよい方法があるかもしれません．興味のある人は，是非考えてみてください．

6.7 データの参照の最適化

ネタが尽きてきた感もありますが，最適化手法の最後に，データの参照の最適化について簡単に説明します．大した話ではないのですが，最初に複数の変数に配列上の同じデータをロードするような場合，人間が見ると "実質同じデータだから何度もロードしなくてよい" とわかるような変数でも，コンパイラからするとそうは見えず，(結果的に) 同じアドレスから何度も値を読み出すコードを生成してしまう可能性があります．ここでもリスト 6.34 のメインループ内のロードの部分を例に考えてみましょう．抜き出したものをリスト 6.42 に示します．

既にお忘れかもしれませんが，k，k1，k2，k3 はそれぞれ 0，1，99，100 から始まる配列のインデックスでした．したがって，多くの場合，k + n + 1 と k1 + n は等しく，k2 + n + 1 と k3 + n は等しいことになります．しかしながら，コンパイラはそういう事情はお構いなしに，

リスト 6.42 従来のデータのロード

```
v1a[0] = vmt[k];
v1a[1] = vmt[k + 1];
v1a[2] = vmt[k + 2];
v1a[3] = vmt[k + 3];

v1b[0] = vmt[k1];
v1b[1] = vmt[k1 + 1];
v1b[2] = vmt[k1 + 2];
if (k1 == 153)
    k1 = -3;
v1b[3] = vmt[k1 + 3];

v2a[0] = vmt[k2];
if (k2 == 155)
    k2 = -1;
v2a[1] = vmt[k2 + 1];
v2a[2] = vmt[k2 + 2];
v2a[3] = vmt[k2 + 3];

v2b[0] = vmt[k3];
v2b[1] = vmt[k3 + 1];
v2b[2] = vmt[k3 + 2];
v2b[3] = vmt[k3 + 3];

if (k3 == 152)
    k3 = -4;
```

これらはインデックスが異なる変数で表されているため，別の変数を参照する可能性があるとみなして，個別にロードするコードを生成します． ロードするデータの半分近くが重複しているというのにこれは非常に勿体ないと考えられます．

そこで，これをリスト 6.43 のように書き換えてみましょう．

リスト 6.43 必ず同じデータを参照する部分は k や k2 を用いるように変更した記述

```
v1a[0] = vmt[k];
v1a[1] = vmt[k + 1];
v1a[2] = vmt[k + 2];
v1a[3] = vmt[k + 3];

v1b[0] = vmt[k + 1]; // k1から変更
v1b[1] = vmt[k + 2]; // k1+1から変更
v1b[2] = vmt[k + 3]; // k1+2から変更
if (k1 == 153)
    k1 = -3;
v1b[3] = vmt[k1 + 3];

v2a[0] = vmt[k2];
if (k2 == 155)
    k2 = -1;
v2a[1] = vmt[k2 + 1];
v2a[2] = vmt[k2 + 2];
v2a[3] = vmt[k2 + 3];

v2b[0] = vmt[k2 + 1]; // k3から変更
v2b[1] = vmt[k2 + 2]; // k3+1から変更
v2b[2] = vmt[k2 + 3]; // k3+2から変更
v2b[3] = vmt[k3 + 3];

if (k3 == 152)
    k3 = -4;
```

k1 だった部分は k + 1 とし，k3 + 1 だった部分は k2 + 2 と書き換えるといったように，複数種類のインデックスで同じデータをロードしている部分を単一のインデックスに統一しています．一見すると元のコードと何一つ変わっていないように見えますが，実際に動かしてみると性

能が改善していることがわかります．測定結果を表 6.14 に示します．

表 6.14 ロードの際に同じ要素を参照する部分のインデックスを揃えた場合の性能改善

状態	実行時間
揃える前	9,066,002
揃えた後	8,707,186

元々奇数パイプに余裕があり，ソフトウェアパイプライニングによってロードの位置が最適化されていたためか，そこまで大きく性能は改善しませんでしたが，それでも数 % 改善されていることは間違いありません．

このように，データの参照のインデックスを工夫してコンパイラにヒントを出すことで，より効率的なコードが生成されるようになる場合があります．たかがロードと思って手を抜かず，人間だからこそ判断できる部分はしっかり最適化させておくべきでしょう．

困ったときの volatile

さらに SPE のプログラムの最適化が進んで，レジスタを目一杯使い，コアループの中のほとんどの部分が 2 命令同時発行となった状態で，あと少し性能を上げたいような場合，適当な配列に `volatile` をつけてみるとコンパイラによる配列のレジスタへの割り付け方の傾向が変化し，(閉塞した状況の中においては) 劇的に性能が向上する場合があります．これは半分 (というよりほぼ完全に) 博打のようなテクニックなのですが，行き詰まった状況を打破する上では意外と効果的だったりします．

6.8 関数のインライン展開

関数のインライン展開は，マクロ関数のように，呼び出し元に直接関数の処理を展開して埋め込む仕組みです．マクロ関数と違って，関数のインライン展開は本物の関数を対象とするため，引数や戻り値の型を設定することができ，誤った型での呼び出しを防止できるなどのメリットがあります．

一般的に，関数をインライン展開することで，関数の呼び出しや関数から戻る際にかかる処理のオーバヘッドを削減することができます．これにより処理の効率化を図ることができます．特に，非常に小さい関数を何度も呼ぶような場合に効果的です．しかしながら，何でもかんでもインライン展開すればその分速くなるかというと，そういうわけでもありません．たとえばインライン展開をせずに，独立した関数として確保した方が，変数へのレジスタの割り付けが効率的に行われるようになり，かえって性能が出る場合があります．また，インライン展開によって関数のブロックが巨大化し，コンパイラが関数全体を満足に最適化できなくなってしまう可能性も考えられます．このように，一概に常にインライン展開すべきであるとは言い切れないのが実状です．

このようなことから，ある関数をインライン展開すべきか否かは，“両方の場合を試してみて，結果を見てから判断する” のが最も確実です．ここでは，これを確実に行えるよう，プログラム中でインライン展開の有無を細かく制御する方法を紹介します．

6.8.1 インライン展開をさせる

GCC は，最適化オプションに-O3 を指定した場合，static な関数について勝手にインライン展開を行おうとする場合があります．また，リスト 6.44 のように，関数の宣言の先頭に“inline”とつけておくと，“この関数をインライン展開して欲しい”ということを GCC に伝えることができます．ただし，この方法ではインライン展開されない場合もあります．

リスト 6.44 関数をインライン展開して欲しいことを伝える記述例

```
inline vec_uint4
genrand_mine_core(int loop)
{
```

常時インライン展開させるには，リスト 6.45 のように，“__attribute__((always_inline))”というコンパイラディレクティブを関数の戻り値の型の後に記述します．

リスト 6.45 関数を常時インライン展開するよう指示する記述例

```
vec_uint4 __attribute__((always_inline))
genrand_mine_core(int loop)
{
```

6.8.2 インライン展開をさせない

逆に，static 属性の関数をインライン展開させたくないという場合もあるでしょう．そのような場合，リスト 6.46 のように，“__attribute__((noinline))”というコンパイラディレクティブを関数の戻り値の型の後に記述することで，インライン展開しないように指示できます．

リスト 6.46 関数をインライン展開しないように指示する記述例

```
static vec_uint4 __attribute__((noinline))
genrand_mine_core(int loop)
{
```

6.8.3 インライン展開の有無を確認する

通常，GCC は，インライン展開の指示が出されている部分について，何らかの理由でインライン展開できないと判断した場合でも，特にメッセージを出すこともなく，まるで最初から指示がなかったかのように通常の関数とその呼び出しを作って済ませてしまいます．そのため，インライン展開を指示したはずなのに，実際にはまったくインライン展開されていなかったといったような事態がよく起こります．これを防ぐには，GCC 実行時に，-Winline というオプションをつけます．このオプションをつけておくと，インライン展開の指示を出した関数がコンパイルの時点でインライン展開できないと判断された場合に警告が出るようになります．

また，実際に生成されたオブジェクトファイルや実行ファイルなどを逆アセンブルすることでも，インライン展開の成否を確認することができます．たとえば逆アセンブルした結果にインライン展開を指示した関数の名前が見当たらなかった場合，その関数はインライン展開されたために呼び出し元に取り込まれたと考えることができます．また，インライン展開を指示した関数名が逆アセンブルした結果に残っている場合でも，呼び出し元でその関数が別途展開されている可

能性もあるので，呼び出し元を探し，そこからインライン展開の指示を出した関数への分岐が発生しているかどうかを確認することで，インライン展開の成否を調べることができます．インライン展開されていたら，呼び出し元にはその関数への分岐が含まれないはずです．

6.8.4 インライン展開されなくて困ったら

何らかの上限を超えてしまったせいで関数をインライン展開できなかったとの警告を受けた場合，`-finline-limit` を使ってインライン展開可能なサイズの上限を上げたり，`--param` オプションで `large-function-growth` を上げて関数のサイズの増大に対する制限を緩和したりすることで，これを回避できる場合があります．筆者はこの辺りは詳しくないので，詳細を知りたい場合は，`spu-gcc` の警告を元に GCC の manpage などで調べてください．

6.9 デクリメンタで実行時間を測る

最後に，デクリメンタで SPE 上のプログラムの実行時間を測定する方法について紹介します．これは最適化の手法そのものではありませんが，処理に要した時間が正しく測れないことにはチューニングの効果を正しく知ることもできませんので，最適化を行う上で必須となる知識と言えるでしょう．これまで実行時間の測定は Hack the Cell 2009 の課題に付属している時間測定機能をそのまま利用していましたが，この時間測定機能も，実はデクリメンタを利用しています．

6.9.1 デクリメンタの読み書き

デクリメンタは各 SPE に搭載されている 32bit 幅のカウンタです．名前が表す通り，一定サイクル ($\neq$ SPE のクロック) で値が 1 ずつ減っていきます．デクリメンタは値の設定や読み出しが簡単にできるようになっているため，プログラムのある範囲の処理にかかっている時間を計測するのに利用できます．たとえば測定したい範囲の直前にデクリメンタに大きな値を設定し，測定したい範囲を抜けた直後にデクリメンタの値を読み出して，最初に設定した値からどれだけ減ったかを求めることで，測定開始から終了までの間にどれだけ時間が経過したのかを知ることができます．

デクリメンタに値を設定する場合は，リスト 6.47 のように記述します．この場合，デクリメンタの値は `0xffffffff` となります．また，デクリメンタの値を読み出すには，リスト 6.48 のように記述します．この場合，`d` にデクリメンタの値が格納されます[*19]．

リスト 6.47 デクリメンタへの値の設定

```
spu_writech(SPU_WrDec, 0xffffffff);
```

リスト 6.48 デクリメンタの値の読み出し

```
unsigned d;
  ...
d = spu_readch(SPU_RdDec);
```

*19 ここで呼んでいる `spu_writech` と `spu_readch` については第 9 章にて詳しく説明します．

6.9.2 デクリメンタの値の減少

デクリメンタの値がどれくらいの時間でどの程度減るのかについては，PS3 Linux 上で`/proc/cpuinfo` を見るとわかります．実際に PS3 Linux 上で`/proc/cpuinfo` を `cat` で読み出した結果を以下に示します．

```
% cat /proc/cpuinfo
processor       : 0
cpu             : Cell Broadband Engine, altivec supported
clock           : 3192.000000MHz
revision        : 5.1 (pvr 0070 0501)

processor       : 1
cpu             : Cell Broadband Engine, altivec supported
clock           : 3192.000000MHz
revision        : 5.1 (pvr 0070 0501)

timebase        : 79800000
platform        : PS3
model           : SonyPS3
```

ここで表示されている `timebase` の値が，デクリメンタが1秒でどれだけ減るのかを示しています．したがって，プログラムの2点で取得したデクリメンタの値の差を `timebase` で割った商が，その2点間の処理に要した時間 (sec) となります．

6.9.3 デクリメンタの使い方の例

デクリメンタを使用して，プログラムの特定区間の処理時間を測定する例をリスト6.49に示します．この例では，`spu_writech` を呼んでデクリメンタに値を設定してから，`spu_readch` を呼んでデクリメンタの値を読み出すまでの間にどれだけデクリメンタが減ったかを調べ，それを `TIMEBASE` で割って時間 (sec) を求めています．

リスト 6.49 デクリメンタを用いて処理時間を測る例

```
#define TIMEBASE 79800000
    volatile int i;
    float loop_time;

    spu_writech(SPU_WrDec, 0xffffffff);

    for (i = 0; i < 400000000; i++)
        ;

    loop_time = (float)(0xffffffff - spu_readch(SPU_RdDec)) / (float)TIMEBASE;
    printf("%f(sec)\n", loop_time);
```

なお，デクリメンタは32bitのカウンタなので，PS3の場合だと54秒弱で1周することになります．リスト6.49の方法はあまり長い時間の測定には向かないという点に注意が必要です．第14章で詳しく紹介しますが，デクリメンタが1周した際にイベントを発生させることが可能なので，イベントを適宜検出すれば，より長い時間の計測を行うことも可能になります．また，当然ですが，デクリメンタの値を読み書きする処理にもそれなりの時間がかかるので，変な使い方をしてしまうと意味のある測定ができなくなってしまいます．注意しましょう．

第 7 章

Linux と SPE

最終回じゃないぞよ
もうちっとだけ続くんじゃ

— 鳥山明『ドラゴンボール』より —

本章では，これまでの SPE 固有のプログラミングの話から少し離れて，Linux 上のユーザプログラムから SPE を利用する際の仕組みについて簡単に紹介します．また，本書で SPE の制御に用いているユーザライブラリである libspe2 が，内部でどのようにして SPE を制御しているのかについても簡単に紹介します．

7.1 spufs

Linux では，SPE の提供する様々なリソースはファイルとして表現され，ユーザはこれらのファイルにアクセスすることで SPE を制御するようになっています．このファイルを用いたリソース管理の仕組みを提供しているのが，Linux カーネルに組み込まれている spufs と呼ばれる擬似ファイルシステムになります．

PS3 Linux 上で `mount` コマンドを実行すると，“`spufs on /spu type spufs (rw)`” という行が表示されるはずです (Fedora 9 の場合)．これは，“spufs の提供する (擬似) ファイルシステムが`/spu` にマウントされている” ということを意味しています．`ls` コマンドで`/`直下を見てみると，“`spu`” というディレクトリがあることが確認できます．spufs の提供する各種擬似ファイルは`/spu` の下に生成されます．

Linux が起動した直後は，`/spu` の下には何のファイルもディレクトリも存在しませんが，Linux カーネルの内部に SPE のコンテキストが生成されると，`/spu` の直下にディレクトリが作られ，その下に SPE の様々なリソースに対応したファイルが生成されます．これらのファイルやディレクトリは，SPE のコンテキストが解放される際に同時に消去されます．なお，ここでのコンテキストは，libspe2 でユーザが `spe_context_create` を呼んだ際に作られるコンテキストとは別の，Linux カーネル内で生成・管理されるコンテキストになります[*1]．

[*1] もっとも `spe_context_create` を呼ぶと libspe2 内部で Linux カーネルの SPE コンテキストを生成する処理が呼ばれることから，これらは非常に密な関係にあると言えます．

7.1.1 Linux カーネル内での SPE コンテキストの生成

spufs は，Linux カーネル内にコンテキストが生成された際に，そのコンテキストにアクセスするための起点となるファイルを生成しますが，コンテキストそのものを生成するための起点はファイルとして提供していません．ユーザプログラムからのコンテキストの生成は，/spu の下のファイルをアクセスするのではなく，spu_create というシステムコールを呼ぶことで行います．libspe2 を利用して SPE プログラミングをする上では必要のない知識かもしれませんが，参考までに紹介しておきます．

spu_create: Linux カーネル内で SPE のコンテキストを生成する

```
int spu_create(const char *pathname, int flags, mode_t mode);
int spu_create(const char *pathname, int flags, mode_t mode,
               int neighbor_fd);
```

spu_create は，Linux システムコールの 1 つで，Linux カーネル内に SPE のコンテキストを生成します．当然ですが，CBEA に準拠したプロセッサ上で動作する PowerPC 向け Linux においてのみ利用することができます．spu_create を呼ぶには sys/types.h と sys/spu.h を include しておく必要があります．

pathname は，生成したコンテキストに対応する spufs のディレクトリを，どこにどのような名前で作るのかを指定するのに用います．pathname には，spufs の提供するディレクトリの下の存在しないディレクトリのパスを指定する必要があるため，spufs が/spu 以下にマウントされている場合，ここでは "/spu/" から始まる文字列を pathname として渡すことになります．spufs は，ここで指定したパスの位置に，コンテキストに対応したディレクトリを生成します．

また，flags はコンテキストの属性を指定するのに用います．通常は 0 を指定しますが，コンテキストのスケジューリングに関連した指示を出す場合などは，ここでフラグを設定します．

mode には，spufs によって/spu に作られるディレクトリやファイルのアクセス権限を指定します．型から想像がつくかもしれませんが，ここには open などで指定するのと同じように，S_IRUSR や S_IWUSR などを OR した値を用います．

neighbor_fd は，他の既存の SPE コンテキストとの間にアフィニティ属性を持たせる場合に特別に指定します．アフィニティ属性は，Linux カーネルが SPE のコンテキストをスケジューリングする際に，指定した別の SPE のコンテキストと同時に実 SPE 上で実行されるように指示する際に用います．本書では扱いませんが，興味のある方は文献 [4] などを参照してみてください．

spu_create は，open などと同じように，戻り値として，生成されたディレクトリに対応するファイルディスクリプタを返します．このファイルディスクリプタは，SPE のコンテキストを実行する際に用いる spu_run システムコールの引数として渡したり，close することでコンテキストを解放するのに用いたりといった，特定の用途に限って用いることができます．エラーの場合は −1 が返り，errno に原因を示すコードが設定されます．

7.1.2 SPE コンテキストの実行

ついでなので，Linux カーネル内の SPE コンテキストの実行についても紹介しておきます．

spu_create で生成した SPE コンテキストのスケジューリングを開始し，実 SPE 上で実行されるようにするには，spu_run というシステムコールを用います．

spu_run: SPE のコンテキストを実行する

```
int spu_run(int fd, unsigned *npc, unsigned *event);
```

spu_run は，spu_create で生成したコンテキストを実行する (物理的に利用可能な SPE に割り付けるようスケジューリングを開始させる) システムコールです．spu_run を呼ぶには sys/spu.h を include する必要があります．

fd に spu_create の戻り値のファイルディスクリプタを渡すことで，どのコンテキストを実行するのかを指示します．

また，npc には，そのコンテキストの実行開始アドレスを保持した変数のポインタを渡します．spu_run は，一度呼ばれると SPE の処理が停止するまで関数から戻らないブロッキング型の関数です．関数から戻った際，npc の指す先の変数に，停止した際に実行していた命令の次に実行する命令のアドレスが格納されます．したがって，プログラムが何らかの事情により中断して spu_run から戻った際に，原因を除去した上で，npc の指す先の値はそのままに再度 spu_run を呼ぶと，中断した状態から SPE の動作を再開させることができます．

event には，SPE が停止した際に，その原因などを格納させたい変数のポインタを指定します．ここには NULL を指定することも可能です．

先ほども述べたように，spu_run は SPE が停止するまで戻らない関数です．停止した際には，戻り値として SPE の状態を示す spu_status と呼ばれるレジスタの値を返します．したがって，戻り値を見ることで，何が原因で SPE が停止したのかを判別することができます．この戻り値は，stop-and-signal を用いた外部関数呼び出しの仕組みなどで利用されています．また，エラーの場合は −1 が返り，errno に原因を示すコードが設定されます．

なお，spu_status の内容については本書では扱いませんが，興味がある場合は文献 [1] などを参照するか，あるいは spu_run の manpage を参照するなどしてください．spu_run の manpage は，コマンドライン上で “man spu_run” と入力することで参照することができます (spu_create についても同様に manpage を参照可能)．最近のディストリビューションであれば，PS3 上にインストールした Linux でなくても spu_run や spu_create の manpage を備えているようです．

7.1.3 spufs の提供するファイル

後でまた詳しく説明しますが，libspe2 の spe_context_create が呼ばれると，その内部で spu_create システムコールが呼ばれて SPE のコンテキストが生成されます．この状態で/spu の下を参照すると，以下のようなディレクトリが生成されているはずです*2．

*2 ディレクトリ名は libspe2 が勝手につけたものなので，spufs の挙動とは直接関係がありません．ちなみに，libspe2 は呼び出し元のプロセスの PID と libspe2 側の SPE コンテキストのアドレスを元にディレクトリ名を決定します．これにより，ディレクトリの名称 (パス名) が衝突することを避けています．

```
% ls /spu
spethread-1791-268509248/
```

このディレクトリの中を見てみると，以下のように多数のファイルが生成されていることが確認できます．これらのファイルは，いずれも所有者が root などではなく自分のユーザアカウントになっています (以下の例では筆者のアカウントである “nosuke:nosuke” となっている)．これらのファイルを read，write，mmap などのシステムコールでアクセスすることで，SPE の各種ハードウェアリソースにアクセスすることができます．libspe2 もこれらファイルを介して SPE の様々なリソースの管理を行っています．

個別のファイルがどのようなものなのかについては，あまりに量が多いためここでは紹介しませんが，これらの説明についても spu_run などと同様に manpage で参照できるようになっているので，興味のある方は “man spufs” とコマンドを入力してみてください．

```
% ls -l /spu/spethread-1791-268509248
合計 0K
-r-------- 1 nosuke nosuke      0 2009-07-16 23:26 capabilities
-rw------- 1 nosuke nosuke      0 2009-07-16 23:26 cntl
-rw------- 1 nosuke nosuke      0 2009-07-16 23:26 decr
...
--w------- 1 nosuke nosuke      0 2009-07-16 23:26 wbox
-r-------- 1 nosuke nosuke      4 2009-07-16 23:26 wbox_info
-r-------- 1 nosuke nosuke      4 2009-07-16 23:26 wbox_stat
```

7.1.4　spufs の実体

spufs は，Linux のカーネルに組み込まれており，ソースコードは Linux カーネルのソースツリーの arch/powerpc/platforms/cell/spufs ディレクトリ以下に配置されています．ソースを読むことで，上記の各種ファイルを mmap したり read したりした際に何が起こるのかを正確に知ることができるでしょう．また，1 つ上がった arch/powerpc/platforms/cell ディレクトリには，先ほど紹介したシステムコールのソースコードなども置かれています．興味のある方は参照してみてください．

7.2　libspe2

libspe2 は，LGPL ライセンスで配布されているオープンソースな SPE の実行環境の 1 つです．libspe2 は PPE 用のライブラリで，第 4 章で紹介したように，ユーザアプリケーションから SPE のプログラムを実行させるのに必要となる API を提供しています．また，それ以外にも SPE を便利に活用するための多数の API を提供しています．これらについては第 9 章以降で順次紹介していきます．

7.2.1 libspe2 と spufs

前節の説明を読んで，libspe2 の API のモデルと spufs に関連するシステムコール (spu_create と spu_run) のモデルはよく似ていると感じたかもしれません．実際のところ，libspe2 の spe_context_create は内部で spu_create を呼んでおり，spe_context_run は内部で spu_run を呼んでいます．一方，spe_context_destroy に相当するシステムコールは用意されていません．spe_context_destroy では spu_create の返してきたファイルディスクリプタを close することで，Linux 側の SPE コンテキストの解放を行います．spe_context_destroy を呼ばずに PPE のプログラムが終了しても特に問題ないのは，ユーザプログラムが終了する際に，開いていたファイルをすべて閉じるようになっているためです．

また，第 9 章で紹介する，SPE との間でのメッセージ通信を行うための API などは，内部で spufs の提供するファイルを read や write などでアクセスすることで機能を実現しています．

7.2.2 libspe2 と MMIO

SPE のレジスタは，Memory-Mapped I/O (MMIO) と呼ばれる仕組みで PPE のアドレス空間上にマップすることができ，これにより PPE から直接アクセスできるようになっています．spufs では，この仕組みを利用して SPE のレジスタをカーネル空間上にマップすることで，SPE の様々なリソースへのアクセスを行います．

spufs の提供する各種ファイルを open して read や write をした場合，Linux カーネル内部では，SPE のコンテキストを実 SPE に割り当てた上で，アドレス空間にマップされた SPE のレジスタに対して読み書きを行います．libspe2 の API を呼び出した場合も，標準では spufs のファイルを read・write することで SPE の制御を行います．

また，MMIO で PPE のアドレス空間にマップ可能なレジスタの一部には，プロブレムステート領域と呼ばれる特殊な領域が用意されています．この領域には，カーネルなどの特権ソフトウェアからだけでなく，ユーザレベルのソフトウェアからも自由に値を設定することが許されているレジスタが集められています．4.2 でも少し触れましたが，spe_context_create の引数に SPE_MAP_PS というフラグを与えた場合，このプロブレムステート領域がユーザプログラムのアドレス空間にもマップされるようになります．その場合，libspe2 は spufs のファイルを read・write せずに，可能な限りアドレス空間にマップされたプロブレムステート領域のレジスタを直接アクセスしようとします．これにより，read や write などのオーバヘッドを削減できることになるため，spe_context_create の引数のフラグには極力 SPE_MAP_PS をセットしておくことをおすすめします．プロブレムステート領域のレジスタの詳細などに関しては，第 12 章で詳しく紹介します．

7.2.3 libspe2 の外部関数呼び出し

4.3.3 でも少し触れましたが，printf などの関数は，SPE 上で完結せず，途中で PPE 上の関数を呼び出すことで実現されています．このような仕組みを “外部関数呼び出し” と呼びます．libspe2 には，printf だけでなく，fopen や fread などの C99 の関数や，dup や rmdir などの POSIX.1 系の関数なども用意されており，多数の標準的な関数を SPE 上から利用できるように

なっています*3.

外部関数呼び出しがどのように実現されているかといった話題や，実際にユーザが外部関数を独自に記述するための方法などについては，第 13 章で詳しく扱います.

SPE のデバッガ

筆者が詳しくないということもあって，本書ではデバッガの話は一切取り上げていませんが，1 点だけ気をつけるべき点を紹介しておきます．PPE 上のプログラムから `spe_context_run` を呼んで実行を開始させた SPE のプログラムをデバッグする場合，うっかり通常の `gdb` コマンドを叩いてデバッガを起動してしまいがちなのですが，ここでは `gdb` ではなく `ppu-gdb` を使うようにしてください．通常の `gdb` だと，`spe_context_run` の先で `spu_run` システムコールを呼んだところまでしか見ることができないのですが，`ppu-gdb` を使うと，SPE 側のプログラムの `main` から先もそのまま覗けるようになります.

*3 SPE からこれらを利用することにどれほどの価値があるのかについてはよくわかりませんが ….

第 8 章
SPE を複数使うプログラム

> クリナップ クリンミセス!!
> 武術によくある専門用語だろうか …？
> 何をやるのかさっぱりわからん …
> — うすた京介『セクシーコマンドー外伝 すごいよ!!マサルさん』より —

第 4 章で紹介した “Hello World” は，PPE のプログラム中で SPE のコンテキストを 1 個だけ生成して実行するだけのプログラム，すなわち 1 個の SPE を動かして終わるだけのプログラムでした．せっかく PS3 Linux では SPE を 6 個まで同時に動かすことができるというのに，これはちょっと寂しい話です．そこで，本章では，複数の SPE を同時に動かす方法を紹介します．

8.1 複数コンテキストの同時実行

複数の SPE を同時に動かす場合でも，API は第 4 章で使ったものと同じものを用います．第 4 章の例では 1 個しか用意しなかった SPE のコンテキストを複数用意し，`spe_context_run` を立て続けに複数回呼んで，すべてのコンテキストを走らせればよいのです．ただし，これには 1 つ問題があります．`spe_context_run` は，実行を開始した SPE プログラムが終了するまでブロックする関数でした．そのため，普通にシーケンシャルなプログラムを書いていては，`spe_context_run` で SPE プログラムの実行を開始した後，その SPE の処理が終わるまで次の `spe_context_run` を呼ぶことができません．ノンブロッキングな `spe_context_run` があればよいのでしょうが，libspe2 にはそのようなものは用意されていません*1．したがって，複数の SPE を同時に動かすには，もう一工夫必要になります．

もったいぶっても仕方ないので，どうすればよいのか書きますと，ここではスレッドを使います．複数の SPE コンテキストを同時に実行させるには，PPE プログラム中で直接 `spe_context_run` を呼ばずに，一旦サブスレッドを生成し，サブスレッドの中で `spe_context_run` を呼び出すことを行います．これにより，SPE を呼び出したサブスレッドはブロックするものの，メインスレッドはブロックしなくなるため，メインスレッドは次のサブスレッドを生成して別の SPE のコンテキストの実行を開始させたり，あるいは現在動作している SPE の状態を監視したりすることが可能となります．

libspe2 では，複数の SPE を動かす際に pthread を用いることを推奨しているようです．

*1 初代 libspe にはあったような気がします．もう忘れちゃったけど …．

pthread を利用してサブスレッドを 3 つ生成し，各々に SPE プログラムの実行を開始させる場合の処理のシーケンスを図 8.1 に示します．図において，“main” と書かれている部分がメインスレッドを表し，“sub0” や “sub1” と書かれている部分が PPE 上のサブスレッドを表します．また，“SPE0” や “SPE1” と書かれている部分は各 SPE 上の処理を示しています．メインスレッドからサブスレッドを複数生成し，サブスレッドの中でコンテキストを走らせることで，サブスレッドがブロックしている間もメインスレッドが動き続けることがわかります．

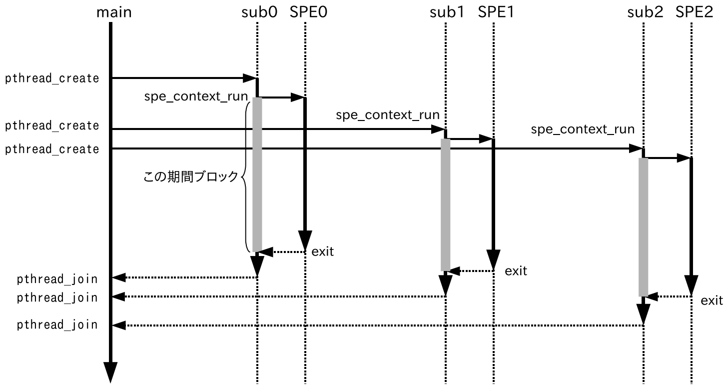

図 8.1　libspe2 での複数 SPE を用いたプログラムを実行する際のシーケンス

8.2　複数の SPE を同時に動かす例

以下では pthread を利用して 6 個の SPE を同時に動かす例を紹介します．まず，PPE 用のプログラムであるリスト 8.1 のコードを入力し，`ppu-gcc` でコンパイルして `hello6_ppe` という実行ファイルを用意します．

次に，SPE 用のプログラムであるリスト 8.2 のコードを入力し，`spu-gcc` でコンパイルして `hello6_spe` という実行ファイルを用意します．

このプログラムは，PPE から 6 個の SPE を起動して，各 SPE で `printf` を行うものになります．動かすと，以下のような出力が出た状態でプログラムが停止するはずです[*2]．

```
% ./hello6_ppe
start SPE programs
Hello World #0
Hello World #1
Hello World #2
Hello World #3
Hello World #4
Hello World #5
```

理由は後で説明しますが，この例の SPE のプログラムの最後には無限ループが入っているため，いつまで待っても SPE のプログラムは終了しません．ひとまず `Ctrl-c` で終了させてください．

[*2] “#” の後ろの番号は，ここで示した例と順番が違うかもしれませんが問題ありません．

リスト 8.1　hello6_ppe.c

```
#include <stdio.h>
#include <pthread.h> // pthreadを使用するのでincludeする
#include <libspe2.h>

// 各スレッドを管理するための構造体
typedef struct spethread_t {
    int id;
    spe_context_ptr_t context;
    pthread_t pthread;
} spethread_t;


// 各スレッドが実行する関数
static void *
spethread_fn(void *arg)
{
    spethread_t *spethread = (spethread_t *)arg;
    spe_context_ptr_t context = spethread->context;
    unsigned entry = SPE_DEFAULT_ENTRY;
    spe_stop_info_t stop_info;
    spe_context_run(context, &entry, 0, (void *)(unsigned long)spethread->id,
                    NULL, &stop_info);
    pthread_exit(NULL);
}

#define SPE_NUM 6

int
main(int argc, char *argv[])
{
    spe_program_handle_t *program;
    spethread_t spethreads[SPE_NUM];
    int i, flags = SPE_MAP_PS;

    program = spe_image_open("hello6_spe");

    // 各SPEの値を生成
    for (i = 0; i < SPE_NUM; i++) {
        spethreads[i].id = i;
        spethreads[i].context = spe_context_create(flags, NULL);
        spe_program_load(spethreads[i].context, program);
    }

    printf("start SPE programs\n");

    // SPEのコンテキストを実行する各スレッドを生成
    for (i = 0; i < SPE_NUM; i++)
        pthread_create(&spethreads[i].pthread, NULL, &spethread_fn,
                       &spethreads[i]);

    // SPEのコンテキストを実行する各スレッドの終了を待ってコンテキストを破棄
    for (i = 0; i < SPE_NUM; i ++) {
        pthread_join(spethreads[i].pthread, NULL);
        spe_context_destroy(spethreads[i].context);
    }

    spe_image_close(program);
    printf("done\n");
    return 0;
}
```

リスト 8.2 **hello6_spe.c**

```
#include <stdio.h>

int
main(unsigned long long spe, unsigned long long argp, unsigned long long envp)
{
    int speid = (int)argp;
    printf("Hello World #%d\n", speid);

    // 複数SPEが確実に同時に動いている状態を作るための無限ループ
    while (1)
        ;
    return 0;
}
```

8.2.1 PPE 側のプログラムの解説

PPE 側のプログラムの基本的な流れは，4.3 で示した SPE 1 個版と同じで，SPE の実行ファイルを読み込んで，SPE のコンテキストを作り，そこにプログラムをロードして，SPE 上で実行するということを行っています．大きな違いは，コンテキスト (を含む構造体) を複数作った上で，`main` 関数の中で直接 SPE プログラムを呼ぶ代わりにスレッドを複数作り，その作った各スレッドの中で SPE のプログラムを実行するようになった点です．

上から見ていきましょう．まず，プログラムの先頭で，リスト 8.3 に示す `spethread_t` 構造体を定義しています．この構造体は，SPE のコンテキストである `spe_context_ptr_t` 型の `context`，pthread のスレッドを管理する `pthread_t` 型の `pthread` および `int` 型の `id` の 3 つのメンバを持っています．このプログラムでは，コンテキストとそれを実行するスレッドをひとまとめにして管理したいがために，このような構造体を導入しています．この構造体は，“libspe2 を利用する場合はこう書かなければならない” というものではなく，単に筆者の好みでこの組み合わせにしています．自分で新たにプログラムを書く際は，適宜使いやすいものを用意するとよいでしょう．

リスト 8.3 spethread_t 構造体の定義

```
typedef struct spethread_t {
    int id;
    spe_context_ptr_t context;
    pthread_t pthread;
} spethread_t;
```

次に，リスト 8.4 に示すように，`spethread_fn` という関数を定義しています．関数の中を見ると `spe_context_run` が呼ばれており，ここで SPE のプログラムが開始することがわかります．`main` で生成される各サブスレッドは，この関数から実行を開始します．`main` 内で呼び出している部分を見るとわかりますが，この関数に引数として渡されている `arg` は，各スレッドが扱う SPE に対応する `spethread_t` 構造体のアドレスになります．

リスト 8.4　spethread_fn の定義

```
static void *
spethread_fn(void *arg)
{
    spethread_t *spethread = (spethread_t *)arg;
    spe_context_ptr_t context = spethread->context;
    unsigned entry = SPE_DEFAULT_ENTRY;
    spe_stop_info_t stop_info;
    spe_context_run(context, &entry, 0, (void *)(unsigned long)spethread->id,
                    NULL, &stop_info);
    pthread_exit(NULL);
}
```

spethread_fn の中では，第 4 章の場合と異なり，spe_context_run の第 4 引数に NULL ではない値を渡しています．この値はプログラム中で定義した SPE の ID になります (ポインタに見せかけて渡している)．SPE のプログラムで printf を呼ぶ際に，この引数の値を出力させることで，どの SPE による出力かを判別できるようにしています．また，最後に pthread_exit を呼んでいますが，ここでは NULL を返しているだけなので，呼ばなくても問題ありません．

最後に main 関数を見てみましょう．SPE_NUM は 6 と定義されているので，適宜読み替えてください．main では，まず，リスト 8.5 に示すように spe_image_open でプログラムをファイルから読み込み (すべての SPE で同じ実行ファイルを利用するので spe_image_open は 1 回だけ呼べばよい)，その後のループで配列として複数確保してある spethread_t 型の構造体のデータの各々に対して spe_context_create でコンテキストを生成して割り当て，spe_program_load でそれらコンテキストにプログラムを割り付けています．また，各 spethread_t 型のデータにコンテキストの識別子として 0 から始まる ID を設定しています．数は増えましたが，中でやっている処理は第 4 章の PPE のプログラムと同じです．

リスト 8.5　プログラムの読み込みから各コンテキストへの割り付けまで

```
    spe_program_handle_t *program;
    spethread_t spethreads[SPE_NUM];
    int flags = SPE_MAP_PS;

    program = spe_image_open("hello6_spe");

    for (i = 0; i < SPE_NUM; i++) {
        spethreads[i].id = i;
        spethreads[i].context = spe_context_create(flags, NULL);
        spe_program_load(spethreads[i].context, program);
    }
```

その次のループでは，リスト 8.6 に示すように，pthread_create を SPE の数と同じ回数呼んでスレッドを生成しています．各スレッドは，前述の spethread_fn から処理を始めます．また，各スレッドの引数として，コンテキストを含む spethread_t 構造体の配列の各要素のポインタを渡しています．これにより，各スレッドで異なる SPE コンテキストが実行されることになります．

リスト 8.6　各コンテキストを制御するスレッドの生成

```
    for (i = 0; i < SPE_NUM; i++)
        pthread_create(&spethreads[i].pthread, NULL, &spethread_fn,
                       &spethreads[i]);
```

3 つ目のループでは，リスト 8.7 に示すように，先ほど作った各スレッドの終了を pthread_join で待った上で，spe_context_destroy を呼んでコンテキストを破棄することを，SPE の数と同じ回数行っています．ただし，今回の例では SPE のプログラムが無限ループするようにしてあるので，メインスレッドは pthread_join を 1 回呼んだところでブロックし，このループを抜けることはありません．プログラム上は，この後 spe_image_close を呼ぶことで実行ファイルを閉じて終了となります．

リスト 8.7 各コンテキストを制御するスレッドの終了待ち

```
for (i = 0; i < SPE_NUM; i ++) {
    pthread_join(spethreads[i].pthread , NULL);
    spe_context_destroy(spethreads[i].context);
}
```

8.2.2 SPE 側のプログラムの解説

今回の SPE のプログラムでは，第 4 章の例と異なり，リスト 8.8 に示すように argp で渡された値を speid という変数にコピーしています．argp には，先ほど PPE のプログラムで spe_context_run の第 4 引数に設定した値が入るため，各 SPE はこれを参照することで自分の ID が何なのかを知ることができます．その後の printf 関数では，異なる SPE が動いていることを確認するために speid の中身を出力させています．

リスト 8.8 main 関数での speid の初期化と出力

```
int speid = (int)argp;
printf("Hello World #%d\n", speid);
```

そのすぐ後の while 文が無限ループとなっているため，このプログラムはこれより先には進みません．無限ループは，実際に 6 個の SPE が同時に並列動作している状況を作るために入れてあります．今回の PPE 側のプログラムでは，順次スレッドが生成され，それらのスレッドが同期を取ったりすることなく好き勝手なタイミングで spe_context_run を呼び出します．したがって，“Hello World” と出力するだけの一瞬で終わる SPE のプログラムを動かしてしまうと，最後の SPE のプログラムの実行が開始するよりも前に，最初に実行された SPE のプログラムが終了してしまい，6 個の SPE が同時に動かないといった状況が生じる可能性があるのです．通常は，これを防ぐために SPE と PPE の間で待ち合わせを行ったりするのですが，それに必要な仕組みをまだ紹介していないため (第 9 章で紹介)，ここでは手を抜いて無限ループで済ませてしまいました．無限ループを外せば，hello6_ppe は正常に終了するようになるでしょう．

また，余談になりますが，一般的な並列プログラムでは，この例の speid のように自身に割り当てられた ID を用いてプログラム内で処理を分岐させ，共通のプログラムを実行しつつも個々が異なる処理を行うようにする手法がよく用いられます．これについては第 11 章で例を紹介します．

複数の SPE でプログラムを動かす方法の説明は以上です．もっと複雑な処理を行う場合でも，基本的に PPE 側で行う SPE の起動の手順はこれと変わることはありません．

第 9 章
チャネルとメッセージング機構

それは聞こえるはずもない 魂の声だった
しかし薄れゆく意識の中 富樫の脳裏には はっきりと聞こえてきた
富士をもゆるがす魂の叫び
男塾名物『大鐘音』である

— 宮下あきら『魁!!男塾』より —

SPE には，SPU が MFC とやりとりしたり，自身の持つ特殊なレジスタの値をアクセスしたりするための，チャネルと呼ばれる仕組みが用意されています．本章では，このチャネルと，それを用いて利用することのできる 2 種類のメッセージング機構について紹介します．

9.1 チャネル

SPE は，チャネルと呼ばれる，I/O 空間に相当するアドレス空間を持っています．チャネルは SPU が同じ SPE 上にある MFC に対してコマンドを発行したり，あるいは SPU 自身の情報を取得したりするための機能を提供します[*1]．**各チャネルはいずれも 32bit 幅で，読み出し専用か書き込み専用のいずれか**となります[*2]．SPU からチャネルを参照するには，専用の命令 (チャネルアクセス命令) を用います．

チャネルの中には，中身がセットされていないときに読み出そうとしたり (読み出し専用の場合)，前に書き込んだ値が処理されないうちに次の値を書き込もうとしたり (書き込み専用の場合) すると，SPU の処理が一時的に中断 (ストール) してしまうものがあります．このようなチャネルをブロッキング型のチャネルと呼びます．ブロッキング型のチャネルにアクセスして SPU がストールしてしまうのを回避するために，各チャネルには，チャネルカウンタと呼ばれる副次的な情報が用意されており，SPU からは専用の命令で読み出して値を得ることができます．各チャネルに対応するチャネルカウンタの値を読み出すことで，その時点でそのチャネルから読み出すことのできるデータの数もしくはそのチャネルに書き込むことのできるデータの数を取得することができます．したがって，あるブロッキング型のチャネルのチャネルカウンタの値が 0 かどうかを見ることで，そのチャネルへのアクセスによる SPU のストールの可能性を事前に検知することができます．

[*1] 6.9 で紹介したデクリメンタの値の初期化や読み出しもチャネルを利用して行っています．

[*2] 読み出し専用のチャネルに値を書き込むこともできますが，書き込んでも何も起こりません．また，逆も然りで，書き込み専用のチャネルの値を読んでも 0 が読み出されるだけで何も起こりません．

9.1.1 チャネルアクセス関数

spu-gcc には，チャネルそのものの読み書きやチャネルカウンタの読み出しを行うためのチャネルアクセス命令に対応した型総称組み込み関数が用意されています．

spu_readch：チャネルから値を読み出す

```
d = spu_readch(c);
```

spu_readch は，引数で指定した番号のチャネルから値を読み出して unsigned 型で返します．上記のフォーマットの場合，c で指定した番号のチャネルから値を読み出して d に格納します．

spu_writech：チャネルに値を書き込む

```
spu_writech(c, a);
```

spu_writech は，引数で指定した番号のチャネルに引数で指定した値を書き込みます．上記のフォーマットの場合，c で指定した番号のチャネルに a の値を書き込みます．a は unsigned 型に対応しています．

spu_readchcnt：チャネルカウンタの値を読む

```
d = spu_readchcnt(c);
```

spu_readchcnt は，引数で指定した番号のチャネルのチャネルカウンタの値を読み出して unsigned 型で返します．上記のフォーマットの場合，c で指定した番号のチャネルのチャネルカウンタの値を読み出して d に格納します．c が読み出し専用のチャネルの番号の場合，d にはその時点でそのチャネルからストールせずに確実に読み出すことのできるデータの個数が格納され，c が書き込み専用のチャネルの場合，d にはその時点でそのチャネルにストールすることなく確実に書き込めるデータの個数が格納されます．

9.1.2 チャネルの機能

チャネルは，SPU 自身の情報を読み書きするためのものと，MFC との間で情報をやりとりするためのものの 2 種類に大別することができます．以下にチャネルを利用してできることを機能ごとに紹介します．詳細なチャネルの使い方については，各機能ごとの解説を参照してください．なお，本章以降で “SPU_Rd” や “MFC_Wr” といった文字列で始まる名称が頻繁に登場しますが，これは spu_intrinsics.h で定義されているマクロ名で，チャネル番号に対応します．名前から想像がつくかとは思いますが，“Rd” とついている方が読み出し専用のチャネルで，“Wr” とついている方が書き込み専用のチャネルであることを示しています．なお，一部のチャネルについては “Wr” や “Rd” で始まらない名称のものもあります．

SPU 自身の情報を読み書きするためのチャネル

- デクリメンタ

  6.9 でも紹介しましたが，デクリメンタは SPE 内部にある 32bit 幅のカウンタです．デクリメンタは一定サイクルで値が減るため，SPE 上のプログラムの 2 点間の処理にかかった時間を計測するのに利用できます．SPU は，デクリメンタ関連のチャネルにアクセスすることで，デクリメンタの現在の値の読み出しや値の変更を行うことが可能です．

- イベント

  SPE にはイベントと呼ばれる機構が用意されており，イベントを通じて非同期に発生するさまざまな事象を検出することができます．また，イベントの発生をトリガとして SPU に割り込みをかけることも可能です．SPU は，イベント関連のチャネルを介してイベントの管理を行うことができます．イベントについては第 14 章にて詳しく紹介します．

- メールボックス

  メールボックスは，SPE と外部との間に設けられた単方向の FIFO で，SPE から外部へメッセージを送るのに用いる送信用のものと，SPE が外からのメッセージを受け取るのに用いる受信用のものが用意されています．SPU は，チャネルを介してメールボックスの読み書きを行うことができます．メールボックスの詳細については本章の 9.2 で説明します．

- シグナル

  シグナルは，外部から SPE に対して通知を行う際に用いる，バッファを介したメッセージング機構です．SPU は，チャネルを介して外部から送られたシグナルを読み出すことができます．シグナルの詳細については本章の 9.3 で説明します．

- その他

  これらの他にも，SPU の設定情報を取得するためのチャネルや，割り込み処理時に利用するレジスタをアクセスするためのチャネルなどが存在しますが，本書では扱いません．

MFC との間のやりとりで用いるチャネル

- MFC マルチソース同期

  MFC マルチソース同期は，PPE や他の SPE などの複数のソースから，ある SPE の LS などに対してあった書き込み処理が完了するのを待つ処理です．SPU は，チャネルを介してこの処理の実行を MFC に要求することができます．MFC マルチソース同期については本書では扱いません．

- MFC コマンド

  MFC は，SPU や外部からの DMA 転送要求などを MFC コマンドと呼ばれるコマンドの形で受け付けます．SPU からは，チャネルを介して MFC に対して MFC コマンドを発行します．MFC コマンド関連のチャネルへのアクセスについては，第 10 章で DMA 転送の説明の際に詳しく紹介します．

- MFC タグ

  MFC は，MFC タグと呼ばれる機構を用いて DMA 転送要求の完了状況などを追跡する機能を備えています．SPU からはチャネルを介して MFC に対してコマンドの完了状況を

問い合わせることができます．MFC タグ関連のチャネルについても第 10 章で扱います．

9.2　メールボックス

メールボックスは，SPE が PPE や他の SPE と手軽にメッセージのやりとりを行うための 32bit 幅の FIFO です．SPE が外部からメッセージを受け取るのに用いるメールボックスは，インバウンドメールボックス (受信メールボックス) と呼ばれ，Cell/B.E. では各 SPE に深さが 4 段のものが 1 本ずつ用意されています．一方，SPE が外部へメッセージを送るのに用いるメールボックスには，アウトバウンドメールボックス (送信メールボックス) とアウトバウンド割り込みメールボックス (割り込み送信メールボックス) の 2 種類があり，Cell/B.E. では各 SPE に深さが 1 段[*3]のものがそれぞれ 1 本ずつ用意されています．なお，割り込み送信メールボックスは，名前からすると割り込み通知専用のように思えますが，libspe2 では，デフォルトでは普通のメールボックスとしてユーザが好きに利用できるようになっています．Cell/B.E. 内のメールボックスの配置を模式的に示したものを図 9.1 に示します．

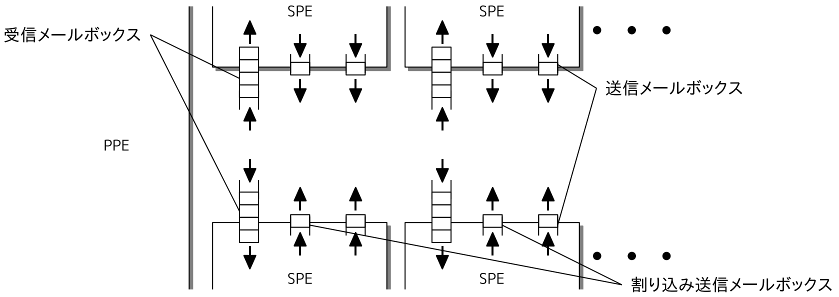

図 9.1　メールボックスの配置

メールボックスは FIFO なので，読み出しが行われないまま FIFO の段数分のメッセージが蓄積されてしまうと，それ以上書き込みを行えなくなってしまいます．また，FIFO であるがゆえに，一度メッセージを読み出すと，そのメッセージは FIFO から消えてしまうので，再度同じメッセージを読み出すことはできません．このような理由から，メールボックスは大量のデータを一気に転送するような用途や，バッファとしての利用には不向きです．相手から何らかの値が書き込まれたことを検出できる点や，事前に PPE と SPE の間で通信領域のアドレスの教え合いなどをしなくてもデータのやり取りが可能な点を活用して，DMA 転送を行うための通信領域のアドレスの通知や，PPE と SPE の待ち合わせなどに用いるのがよいでしょう．

9.2.1　SPE からのメールボックスの利用

SPE は，チャネルをアクセスすることで，自身のメールボックスを読み書きすることができます．メールボックスを利用するためのチャネルの一覧を表 9.1 に示します．

[*3] これを FIFO と呼んでよいのか微妙ですが ….

表 9.1 メールボックスを利用するためのチャネル一覧

チャネル名	R/W	深さ	ブロック	用途
SPU_RdInMbox	R	4	する	受信メールボックス読み出し
SPU_WrOutMbox	W	1	する	送信メールボックス書き込み
SPU_WrOutIntrMbox	W	1	する	割り込み送信メールボックス書き込み

SPU_WrOutMbox チャネルに spu_writech で書き込みを行うと，書き込んだ値をメッセージとして送信メールボックスに格納することができます．また，SPU_WrOutIntrMbox チャネルに関しても同様で，spu_writech で書き込んだ値をメッセージとして割り込み送信メールボックスに格納することができます．一方，SPU_RdInMbox チャネルから spu_readch で読み出しを行うと，受信メールボックスに格納されている外部からのメッセージを取り出すことができます．

SPU_RdInMbox はブロッキング型のチャネルなので，受信メールボックスに何のメッセージも入っていない状態で読み出そうとすると SPU がストールします．また，SPU_WrOutMbox と SPU_WrOutIntrMbox も同様にブロッキング型のチャネルであるため，送信メールボックスや割り込み送信メールボックスの中身がいっぱいの状態でメッセージを書き込もうとした場合，SPU がストールします．ストールを回避したい場合は，事前にチャネルカウンタを読んで，値が 0 でないことを確認しておく必要があります．

9.2.2 PPE からのメールボックスの利用

libspe2 にはメールボックスをアクセスするための API が用意されています．以下で簡単に紹介します．より詳細な情報を得たい場合は文献 [4] を参照してください．

spe_in_mbox_write：受信メールボックスに書き込む

```
int spe_in_mbox_write(spe_context_ptr_t spe, unsigned *mbox_data, int count,
                      unsigned behavior);
```

spe_in_mbox_write は，spe で指定した SPE のコンテキストの受信メールボックスに対して，mbox_data から始まる unsigned 型の配列から，count で指定した個数分のデータを書き込みます．関数をブロックさせるかどうかは，behavior で指定することができます．behavior に指定できるモードを表 9.2 に示します．関数の戻り値は，書き込むことのできたメッセージの個数で，エラーの場合は -1 が返って errno にエラーコードがセットされます．

表 9.2 spe_in_mbox_write で指定可能なブロックモード

モード名	ブロックの仕方
SPE_MBOX_ALL_BLOCKING	指定した個数分書けるまで戻らない
SPE_MBOX_ANY_BLOCKING	メッセージが 1 個以上書けたら戻る
SPE_MBOX_ANY_NONBLOCKING	メッセージが 1 個も書けなくても即戻る

spe_out_mbox_read：送信メールボックスから読み出す

```
int spe_out_mbox_read(spe_context_ptr_t spe, unsigned *mbox_data, int count);
```

spe_out_mbox_read は，spe で指定した SPE のコンテキストの送信メールボックスから，count で指定した個数分のデータを，mbox_data から始まる unsigned 型の配列に読み出します．spe_in_mbox_write と異なり，どこまで読めたら関数から戻るかを指定することはできず，最初から送信メールボックスにメッセージがなかった場合や，count に達する前にメールボックスから読み出せるメッセージが尽きた場合は即座に戻る，ノンブロッキング型の挙動となります．戻り値は，読み出すことのできたメッセージの個数で，エラーの場合は −1 が返って errno にエラーコードがセットされます．

spe_out_intr_mbox_read：割り込み送信メールボックスから読み出す

```
int spe_out_intr_mbox_read(spe_context_ptr_t spe, unsigned *mbox_data,
                           int count, unsigned behavior);
```

spe_out_intr_mbox_read は，spe で指定した SPE のコンテキストの割り込み送信メールボックスから，count で指定した個数分のデータを，mbox_data から始まる unsigned 型の配列に読み出します．spe_in_mbox_write と同様に，関数をブロックさせるかどうかは behavior で指定することができます．behavior に指定できるモードを表 9.3 に示します．戻り値は，spe_out_mbox_read と同様に，読み出すことのできたメッセージの個数となり，エラーの場合は −1 が返って errno にエラーコードがセットされます．

表 9.3 spe_out_intr_mbox_read で指定可能なブロックモード

モード名	ブロックの仕方
SPE_MBOX_ALL_BLOCKING	指定した数分読めるまで戻らない
SPE_MBOX_ANY_BLOCKING	メッセージが 1 個以上読めたら戻る
SPE_MBOX_ANY_NONBLOCKING	メッセージが 1 個も読めなくても即戻る

spe_in_mbox_status：受信メールボックスの FIFO の空きを得る

```
int spe_in_mbox_status(spe_context_ptr_t spe);
```

spe_in_mbox_status は，spe で指定したコンテキストの SPE の受信メールボックスの FIFO に書き込むことのできるメッセージ数を返します．エラーの場合は −1 が返り，errno にエラーコードがセットされます．

spe_out_mbox_status：送信メールボックスのメッセージ数を得る

```
int spe_out_mbox_status(spe_context_ptr_t spe);
```

spe_out_mbox_status は，spe で指定したコンテキストの SPE の送信メールボックスに格納されているメッセージの数を返します．エラーの場合は −1 が返り，errno にエラーコードが

セットされます.

spe_out_intr_mbox_status : 割り込み送信メールボックスのメッセージ数を得る

```
int spe_out_intr_mbox_status(spe_context_ptr_t spe);
```

spe_out_intr_mbox_status は，spe で指定したコンテキストの SPE の割り込み送信メールボックスに格納されているメッセージの数を返します．エラーの場合は −1 が返り，errno にエラーコードがセットされます.

9.2.3 メールボックスの使い方の例

以下でメールボックスの使い方の例をいくつか紹介します．まず，各 SPE が送信メールボックスに値を格納し，それを PPE が読み出す処理の例を示します．この場合，PPE のコードはリスト 9.1 のようになります．この例では，PPE が各 SPE の送信メールボックスにメッセージが 1 つ格納されるのを，spe_out_mbox_read を繰り返し呼ぶことで待ち，メッセージを読み出すことができたらそれを出力しています.

リスト 9.1 PPE による送信メールボックスの読み出し

```
for (i = 0; i < SPE_NUM; i++) {
    // 送信メールボックスに値が書かれるのをポーリングして待つ
    while (spe_out_mbox_read(spethreads[i].context, &mbox_data, 1) < 1)
        ;
    printf("SPE%d: 0x%08x\n", i, mbox_data);
}
```

これに対する SPE のコードはリスト 9.2 のようになります．ここではチャネルカウンタを参照せずに，いきなり SPU_WrOutMbox チャネルに値を書き込んでいます．一見すると各 SPE からの単なる値の受け渡しにしか見えませんが，すべての SPE が送信メールボックスに何かの値を書き込まない限り PPE はリスト 9.1 のループから先に進めないことから，このコードはメッセージのやりとりと同時に，SPE と PPE の “待ち合わせ” を実現していることになります[*4].

リスト 9.2 SPE による送信メールボックスへの書き込み

```
// 送信メールボックスに0x0cbe0cbeを書き込み
d = 0x0cbe0cbe;
spu_writech(SPU_WrOutMbox, d);
```

次に，逆に PPE が各 SPE の受信メールボックスにメッセージを格納し，各 SPE がそれを読み出す処理の例を示します．先に PPE のコードをリスト 9.3 に示します．ここでは，PPE が spe_in_mbox_write を呼んで，mbox_data の値をメッセージとして各 SPE の受信メールボックスに 1 つずつ格納しています．モードが SPE_MBOX_ALL_BLOCKING となっていることから，受信メールボックスがいっぱいの場合でもメッセージを書き終えるまでは関数から戻らない動作となります.

[*4] 細かいことを言うと，この方法ではメールボックスを PPE がポーリングすることになるため，バスに対して大きな負荷となってしまうことが懸念されます．真面目にやるなら主記憶に対する DMA 書き込みで待ち合わせを行うようにすべきでしょう．その場合，DMA 転送による SPE からの書き込みが発生するまでは，PPE はキャッシュからの読み出しを繰り返すことになるので，バスへの負荷は軽減されるはずです.

リスト 9.3 PPE による受信メールボックスへの書き込み

```
mbox_data = 0xce11ce11;
for (i = 0; i < SPE_NUM; i++)
    // 受信メールボックスに値を書き込む (値が書けるまで戻らない)
    spe_in_mbox_write(spethreads[i].context, &mbox_data, 1,
                      SPE_MBOX_ALL_BLOCKING);
```

これに対して，SPE はリスト 9.4 のように `SPU_RdInMbox` チャネルにアクセスして値を読み出そうとします．PPE が受信メールボックスに何かを書くまでは，SPE はリスト 9.4 の行でストールします．受信メールボックスに値が書かれると，SPE は `d` に受信メールボックスの値を読み出した上で先に進みます．よって，この方法を用いることで，メッセージの受け渡しを行いつつ，さらに SPE が PPE を無視してどんどん先に進んでしまうのを防ぐことができます[*5]．

リスト 9.4 SPE による受信メールボックスの読み出し

```
// 受信メールボックスを読み出し (何か書かれるまでストールする)
d = spu_readch(SPU_RdInMbox);
```

9.2.4 チャネルカウンタを使う例

既に述べたように，`SPU_RdInMbox` や `SPU_WrOutMbox` などはブロッキング型のチャネルであるため，メールボックスの状態によってはアクセスの際に SPU がストールしてしまう可能性があります．通常，それで困ることはあまりないのですが，チャネルにアクセスできるようになるまでの期間に他の処理を行いたいなどの理由で SPU がストールするのを避けたい場合は，いきなりチャネルにアクセスせずに，事前にそのチャネルのチャネルカウンタの値を `spu_readchcnt` で取得して，値が 1 以上であることを確認することを行います．チャネルをアクセスする前にチャネルカウンタをアクセスする例として，`SPU_RdInMbox` チャネルについて，チャネルカウンタを参照して値が入ったことを確認してからチャネルそのものを読み出す処理をリスト 9.5 に示します．

リスト 9.5 spu_readchcnt によるチャネルカウンタの読み出しの例

```
// 受信メールボックスのチャネルカウンタが1以上になるまでループ
while (spu_readchcnt(SPU_RdInMbox) < 1) {
    ...
}
// ストールすることなく即座に受信メールボックスの値が読める
d = spu_readch(SPU_RdInMbox);
```

`while` の中で別の処理を進めることができるのであれば，リスト 9.5 のような実装とすると効率がよいでしょう．なお，文献 [1] によると，チャネルカウンタをただひたすらポーリングしてチャネルが読み書きできる状態になるのを待つよりも，チャネルカウンタの値が 0 の状態のチャネルに突撃して SPU をストールさせた方が，待っている間の消費電力を低減できるようなので，特に `while` の中でやるべき処理がないような場合は，チャネルカウンタの値に関わらずアクセスした方がよいのかもしれません．

[*5] こちらの場合は SPU がストールするだけで，主記憶や MMIO へのアクセスが頻発したりすることはないので，バスなどへの負荷は小さいものと考えられます．

9.2.5　割り込み送信メールボックスから割り込みを受ける

メールボックスに関する話題の最後に，割り込み送信メールボックスから実際に割り込みを受け付ける方法を紹介します．libspe2 を用いて割り込み送信メールボックスからの割り込みを受け付けるには，libspe2 のイベント処理を利用します．libspe2 のイベント処理は，一般的なシグナルハンドラなどと異なり，以下のようにして使います．

1. イベントハンドラを作る
2. イベントハンドラにどの SPE コンテキストの何のイベントを待ち受けるかを登録する
3. イベントハンドラにイベントが発生するのを待つ
4. どのイベントが発生したかを調べて適切な処理を行う

ここでいうイベントハンドラは，名前からして，あるイベントが発生した際に呼ばれるコールバック関数かと思われがちですが，そうではなく，複数のイベントをまとめて管理するための特殊なデータ構造になります．libspe2 では，このように，select でファイルディスクリプタのセットを見張るのと似たような感じで，イベントハンドラに登録された複数のイベントを寝て待つことを行います．以下にイベントに関連する libspe2 の API を示します．

spe_event_handler_create : イベントハンドラを生成する

```
spe_event_handler_ptr_t spe_event_handler_create(void);
```

spe_event_handler_create は，イベントハンドラを生成して返します．ここで得られたイベントハンドラに対して，どんなイベントを見張るのかを順次登録していきます．エラーの場合は NULL が返って errno にエラーコードがセットされます．

spe_event_handler_destroy : イベントハンドラを破棄する

```
int spe_event_handler_destroy(spe_event_handler_ptr_t evhandler);
```

spe_event_handler_destroy は，spe_event_handler_create で生成したイベントハンドラを破棄します．evhandler で破棄するイベントハンドラを指定します．本関数は成功したら 0 を返し，エラーの場合は -1 が返って errno にエラーコードがセットされます．

spe_event_handler_register : イベントハンドラの監視対象にイベントを登録する

```
int spe_event_handler_register(spe_event_handler_ptr_t evhandler,
                               spe_event_unit_t *event);
```

spe_event_handler_register は，evhandler で指定したイベントハンドラに対して監視対象とするイベントを登録します．監視対象とするイベントは spe_event_unit_t 型の構造体で定義し，このポインタを渡すことで登録を行います．spe_event_unit_t 構造体と，そのメンバに含まれる spe_event_data_t 共用体の定義をリスト 9.6 に示します．

リスト 9.6 spe_event_unit_t 構造体と spe_event_data_t 共用体の定義

```
typedef struct spe_event_unit {
    unsigned events;
    spe_context_ptr_t spe;
    spe_event_data_t data;
} spe_event_unit_t;

typedef union spe_event_data {
    void *ptr;
    unsigned u32;
    unsigned long long u64;
} spe_event_data_t;
```

監視対象となるイベントを定義するには，まず spe_event_unit_t 構造体の spe メンバにイベントの発生源となる SPE のコンテキストを設定し，events メンバにその SPE で監視したいイベントを OR して指定します．指定可能なイベントを表 9.4 に示します．送信メールボックスへのメッセージの書き込みで割り込みを受ける場合は，SPE_EVENT_OUT_INTR_MBOX というイベントを events メンバに設定します．

表 9.4 イベントハンドラで監視可能なイベント

イベント	発生条件
SPE_EVENT_OUT_INTR_MBOX	割り込み送信メールボックスにメッセージが書かれた
SPE_EVENT_IN_MBOX	受信メールボックスに空きがある
SPE_EVENT_TAG_GROUP	MFC に発行した DMA 転送要求の処理が完了した
SPE_EVENT_SPE_STOPPED	SPE が停止した
SPE_EVENT_ALL_EVENTS	上記のいずれか

data メンバは自由に使うことのできるデータ領域となっています．ここには 8Byte までのデータを指定でき，何を指定しても構わないのですが，後でイベントが発生した際に，この data メンバで指定した値を受け取ることができるので，SPE のコンテキストにつけている番号などを指定しておくと何かと便利です．なお，この関数では，監視対象とするイベントの情報をポインタで渡していますが，実際に呼んだ先でポインタの値が保持されたり，ポインタの指す先の値が変更されたりすることはありません．本関数は成功すると 0 を返し，エラーの場合は −1 が返って errno にエラーコードがセットされます．

spe_event_handler_deregister：イベントハンドラの監視対象からイベントを削除する

```
int spe_event_handler_deregister(spe_event_handler_ptr_t evhandler,
                                 spe_event_unit_t *event);
```

spe_event_handler_deregister は，evhandler で指定したイベントハンドラから監視対象として登録したイベントを削除します．削除するイベントは spe_event_unit_t 型の構造体で定義し，このポインタを渡すことで指定します．削除するイベントをポインタを使って指定していますが，ここで指定する event が spe_event_handler_register で登録した際に用いた spe_event_unit_t 型のデータを指している必要はまったくありません．新規に spe_event_unit_t 型の変数を用意し，そのメンバに適切な値を設定した上で，そのポインタを引数として渡せば，対応するイベントが監視対象から外れます．本関数は成功すると 0 を返

し，エラーの場合は –1 が返って errno にエラーコードがセットされます.

spe_event_wait：イベントハンドラのイベントを待つ

```
int spe_event_wait(spe_event_handler_ptr_t evhandler,
                   spe_event_unit_t *events, int max_events, int timeout);
```

spe_event_wait は，evhandler で指定したイベントハンドラのイベントが発生するまでブロックします．events には，spe_event_unit_t 型の構造体の配列を渡します．イベントが発生すると，events で渡した配列に，発生したイベントに関する情報が設定され，関数から戻ります．events で渡した配列の各要素の spe メンバには，イベントが発生した SPE のコンテキストへのポインタが格納されており，data メンバには監視対象として登録した際に設定していた data メンバの値がそのまま格納されています．また，events メンバには，登録した際に OR したイベントのうち，実際に発生したイベント (表 9.4 で示したイベント) が設定されています.

max_events には，本関数を 1 回呼び出した際に最大何個までのイベントをまとめて受け取るかを指定します．ここには events で指定した配列の要素数以下の数を指定する必要がありますが，本関数は，ここで指定した数分のイベントが検出されるまで待つわけではなく，1 個でも検出できれば戻ります．最後の timeout には，イベントを待つ際のタイムアウト時間を指定します．単位はミリ秒です．–1 を指定すると，少なくとも 1 個はイベントが検出されるまで戻らなくなります．また，0 を指定した場合は，イベントが 1 個も検出できなかった場合でも即座に戻るようになります.

本関数は，成功した場合に発生を検出したイベントの数を返します．したがって，本関数の戻り値の数分の events の要素に有効な値が格納されていることになります．タイムアウトした場合は 0 が返り，エラーの場合は –1 が返って errno にエラーコードがセットされます.

割り込み送信メールボックスからのイベントを待つ

libspe2 の API を用いて，割り込み送信メールボックスに値が書かれた際に発生するイベントを待ち受けるには，spe_context_create の第 1 引数に SPE_EVENTS_ENABLE フラグをセットしておく必要があります．これをやっておかないと，spe_event_handler_register 関数の呼び出しに失敗します.

割り込み送信メールボックスに値が書かれた際に発生するイベントを待ち受ける記述の例をリスト 9.7 に示します．ここでの処理の裏で，各 SPE は 1 回ずつ割り込み送信メールボックスにメッセージを書き込んでいるものとします．したがって，PPE 側では SPE のコンテキスト数分のイベントが発生することになります.

ここでは，最初に spe_event_handler_create でイベントハンドラを生成し，1 つ目のループでこれに各 SPE の SPE_EVENT_OUT_INTR_MBOX を順番に登録しています．次のループの中では spe_event_wait を呼んでイベントを待ち，発生を検出したイベントの数だけイベントの対処 (イベントを発生させた SPE の割り込み送信メールボックスからのメッセージの読み出し) を繰り返します．この後は，イベントハンドラからすべてのイベントを削除して，最後にイベントハンドラ自体を破棄して終了しています.

リスト 9.7 割り込み送信メールボックスからのイベントを待つ処理の例

```
unsigned mbox_data;
// SPEからのイベントを扱う場合はSPE_EVENTS_ENABLEが必要
int flags = SPE_MAP_PS | SPE_EVENTS_ENABLE;
spe_event_handler_ptr_t evhandler; // イベントハンドラ
spe_event_unit_t events[SPE_NUM];  // イベントを受け取る箱

// いつも通りプログラムをロードしてコンテキストを生成して別スレッドで実行
...

// イベントハンドラを生成する
evhandler = spe_event_handler_create();

// 各SPEのイベントを監視する
for (i = 0; i < SPE_NUM; i++) {
    spe_event_unit_t event_tmp;
    // 監視対象は割り込み送信メールボックスのみ
    event_tmp.events = SPE_EVENT_OUT_INTR_MBOX;
    event_tmp.spe = spethreads[i].context;
    // データにはSPEのIDを入れておく
    event_tmp.data.u32 = i;
    // 第2引数はポインタを渡しているが，このように使い回しても問題ない
    if (spe_event_handler_register(evhandler, &event_tmp) < 0) {
        perror("register");
    }
}

// SPEの個数分のイベントを待ち受ける
i = SPE_NUM;
while (i > 0) {
    int rc;
    // イベントをi個，タイムアウトなしで待つ（戻り値が受け取ったイベント数）
    rc = spe_event_wait(evhandler, events, i, -1);
    for (j = 0; j < rc; j++) {
        // イベントが発生したので対応するSPEのメールボックスを読む
        spe_out_intr_mbox_read(events[j].spe,
                               &mbox_data, 1, SPE_MBOX_ALL_BLOCKING);
        printf("SPE%d: 0x%08x\n", events[j].data.u32, mbox_data);
    }
    // 残りのSPEが発生するイベントを待つ
    i -= rc;
}

// 各SPEのイベントの監視を解除
for (i = 0; i < SPE_NUM; i++) {
    spe_event_unit_t event_tmp;
    event_tmp.events = SPE_EVENT_OUT_INTR_MBOX;
    event_tmp.spe = spethreads[i].context;
    spe_event_handler_deregister(evhandler, &event_tmp);
}

// イベントハンドラを破棄する
spe_event_handler_destroy(evhandler);
```

やたらと手間がかかった割には大したことができていませんが，このようにすることで割り込み送信メールボックスの本領を発揮させることができます．

9.3 シグナル

シグナルは，32bit 幅のデータを外部から SPE に通知するのに用いる仕組みです．単方向のメッセージング機構である点が受信メールボックスと似ていますが，FIFO ではないため，SPU が未読の値を PPE や他の SPE が書き潰すことができます．また，PPE 側から設定を行うことで，シグナルに対して値が連続して書かれた際に，後から書かれた値で書き潰すのではなく，元々の値に OR をして新しい値を重ねていくモード (OR モード) に設定することが可能です．

各 SPE にはシグナルを受け取るためのバッファが 2 つずつ用意されています．ここではそれら 2 つを “シグナル 1” および “シグナル 2” と呼び分けます．

シグナルは，外側から見ると単なるバッファ (もしくは書いた値が OR されて蓄積されていく特殊なバッファ) でしかありませんが，SPU が 1 回読み出すと，そのシグナルはゼロクリアされて，何も書き込まれていないときと同じ状態に戻るため，SPE 側からすると FIFO と同じ振る舞いとなります．また，外からはシグナルに何度も続けて値を書き込むことができますが，SPU からは最後に書き込まれた値 (もしくは最後に読み出してから今に至るまでに書き込まれたすべての値の OR) を 1 回だけしか読み出すことができないことから，何度値が書き込まれたかを知ることができません．

9.3.1 SPE からのシグナルの利用

SPE は，自身のシグナルに書き込まれた値を，チャネルをアクセスすることで読み出すことができます．シグナルを利用するためのチャネルの一覧を表 9.5 に示します．

表 9.5 シグナルを利用するためのチャネル一覧

チャネル名	R/W	深さ	ブロック	用途
`SPU_RdSigNotify1`	R	1	する	シグナル 1 読み出し
`SPU_RdSigNotify2`	R	1	する	シグナル 2 読み出し

`SPU_RdSigNotify1` チャネルを `spu_readch` で読み出すと，シグナル 1 に書かれた値を取得できます．`SPU_RdSigNotify2` チャネルに関しても同様です．どちらのチャネルについても，何も値が格納されていない状態で SPU から読み出しを行った場合，SPU はチャネルに読み出し可能な値が格納されるまでストールします．ストールを回避したい場合は，事前にチャネルカウンタを読んで，チャネルカウンタの値が 0 でないことを確認しておく必要があります．

9.3.2 PPE からのシグナルの利用

libspe2 にはシグナルに値を書き込むための API が用意されています．以下で簡単に紹介します．より詳細な情報を得たい場合は文献 [4] を参照してください．

`spe_signal_write`：シグナルを書く

```
int spe_signal_write(spe_context_ptr_t spe, unsigned signal_reg,
                     unsigned data);
```

`spe_signal_write` は，`spe` で指定した SPE のコンテキストの `signal_reg` で示されるシグナルに対して `data` の値を書き込みます．`signal_reg` に指定可能なシグナルのマクロ名を表 9.6 に示します．シグナルの書き込みに成功した場合は 0 が返り，エラーの場合は −1 が返って `errno` にエラーコードがセットされます．

9.3.3 シグナルのモードの切り替え

前述の通り，各シグナルには，書かれた値を OR して蓄積していく OR モードが用意されています．libspe2 では，`spe_context_create` で SPE のコンテキストを生成する際に，どちら

のモードでシグナルを利用するのかを指定します[*6]．具体的には，spe_context_create の第 1 引数に渡すフラグに SPE_CFG_SIGNOTIFY1_OR フラグを OR しておくと，そのコンテキストのシグナル 1 への書き込みが OR モードとなります．同様に，SPE_CFG_SIGNOTIFY2_OR フラグを OR すると，そのコンテキストのシグナル 2 の方に対する書き込みが OR モードとなります．

表 9.6 spe_signal_write で signal_reg に指定可能な値

マクロ名	対応するシグナル
SPE_SIG_NOTIFY_REG_1	シグナル 1
SPE_SIG_NOTIFY_REG_2	シグナル 2

9.3.4 シグナルの使い方の例

以下にシグナルを経由したデータのやりとりの例を示します．まず，PPE のコードをリスト 9.8 に示します．この例で，PPE は各 SPE のシグナル 1 とシグナル 2 にそれぞれ 1 回ずつメッセージを書き込んでいます．

リスト 9.8 PPE によるシグナル 1 およびシグナル 2 への書き込み

```
for (i = 0; i < SPE_NUM; i++) {
    // 各SPEのシグナル1に値を書く
    spe_signal_write(spethreads[i].context, SPE_SIG_NOTIFY_REG_1,
                     0x01020408);
    // 各SPEのシグナル2に値を書く
    spe_signal_write(spethreads[i].context, SPE_SIG_NOTIFY_REG_2,
                     0xa0b0c0d0);
}
```

これに対する SPE のコードはリスト 9.9 のようになります．PPE から各シグナルに書き込みが行われた時点で，SPU_RdSigNotify1 チャネルおよび SPU_RdSigNotify2 チャネルに値が格納され，それらが sig1 および sig2 に読み出されることになります．

リスト 9.9 SPE によるシグナル 1 およびシグナル 2 の読み出し

```
// シグナル1とシグナル2から値を読み出す
sig1 = spu_readch(SPU_RdSigNotify1);
sig2 = spu_readch(SPU_RdSigNotify2);
```

また，シグナルのモードを OR モードに変更する際の記述例をリスト 9.10 に示します．リスト 9.10 では flags に SPE_CFG_SIGNOTIFY1_OR を OR していることから，シグナル 1 のみが OR モードとなります．

リスト 9.10 シグナル 1 を OR モードに設定する際の spe_context_create の例

```
for (i = 0; i < SPE_NUM; i++) {
    spethreads[i].id = i;
    // シグナル1のみORモードにしてコンテキストを生成
    flags = SPE_MAP_PS | SPE_CFG_SIGNOTIFY1_OR;
    spethreads[i].context = spe_context_create(flags, NULL);
    spe_program_load(spethreads[i].context, program);
}
```

[*6] 後から変更する方法はないみたいです …．

第 10 章
DMA 転送

「ブッ殺す」と心の中で思ったならッ！その時スデに行動は終わっているんだッ！
— 荒木飛呂彦『ジョジョの奇妙な冒険』より —

本章では SPE の DMA 転送について紹介します．DMA 転送は，PPE や複数の SPE が連携して動作する本格的なプログラミングを行う上で欠かすことのできない機能です．どのようにすれば何ができるのかということを是非押えておきましょう．

10.1　DMA 転送

一般に，DMA 転送とは，プロセッサが外部のコントローラに対してメモリのアドレスと転送データサイズを通知することで，そのコントローラがプロセッサの処理の裏でデータの転送を行う機構を指します．

SPE では，MFC が DMA コントローラとしての機能を備えており，同じ SPE 上の LS と外部の EA 空間にマップされた記憶領域 (XDR など) との間でのデータ転送を提供します．EA 空間上の，実アドレスに変換可能なアドレスであれば，相手は主記憶 (XDR) である必要はなく，EA 空間にマップされた他の SPE の LS やプロブレムステート領域に対しても DMA 転送でアクセスすることができます[*1]．また，DMA 転送とは言え，間に MMU が介在しているため，メモリ保護などはしっかり行われます．おかしなアドレスへのアクセスは阻止されます[*2]．

MFC に対する DMA 転送の指示は，MFC に対してユーザプログラムが明示的にコマンドを発行することで行います．ハードウェア制御のキャッシュのように，ハードウェアがデータの不足を検知して自動的に DMA 転送の要求を発行したりするような仕組みではありません．

10.1.1　DMA 転送に必要な情報と基本の流れ

MFC は様々な種類の DMA 転送に対応していますが，基本となる DMA 転送において MFC が要求する情報は以下の 5 つになります．

- LS 上の DMA 転送対象領域の先頭アドレス
- EA 空間上の DMA 転送対象領域の先頭アドレス
- DMA 転送サイズ

[*1] EA 空間側の領域として EA 空間にマップされた自身の LS の EA を指定することで，同じ LS 間で DMA 転送を行うことも (制限つきですが) 可能です．
[*2] 何もケアしていないと PPE のプログラムごと終了してしまいます．

- DMA 転送方向 (LS から EA 空間 (PUT) or EA 空間から LS (GET))
- タグ

DMA 転送対象領域のアドレスと転送サイズおよび転送方向については説明するまでもないでしょう．これら 4 つの情報があれば，“どこからどこへどれだけのデータを転送するか” を指定することができます．最後のタグ (MFC タグ) は，DMA 転送要求 (厳密には MFC コマンド) に付随する番号になります．後で詳しく述べますが，タグは 0 から 31 までの値を取ることができ，MFC に対して発行した DMA 転送要求の完了状況の確認や，MFC 内での DMA 転送要求の処理の順番の制御などに用います．また，同じタグのついた DMA 転送要求の集合を “同じタググループに属する DMA 転送要求” などと表現したりします．

SPU から見た DMA 転送の流れは基本的に以下のようになります．

1. MFC に対して DMA 転送の発行に必要な情報を伝え，DMA 転送を要求する
2. MFC に対して DMA 転送処理の完了状況を問い合わせる
3. MFC から DMA 転送処理の完了状況を受け取る

2 で DMA 転送処理の完了状況を問い合わせないと，1 で発行した DMA 転送が完了したのかどうかわからない点に注意してください．DMA 転送が完了したかどうかがわからないと，LS 上の転送領域のデータが更新されているのかどうか (EA 空間からデータを読み出す指示を出した場合) や，LS 上の転送元のデータを書き潰してよいのかどうか (EA 空間にデータを書き込む指示を出した場合) といったことの判断がつかないため，何らかの理由により転送が完了したことが自明である場合を除いて，最終的に処理状況を確認して DMA 転送が完了したことを知る必要があります．なお，ここで挙げた MFC とのやりとりは，すべてチャネルを介して行われます．

10.1.2 DMA 転送の制約

MFC への DMA 転送の要求の方法を示す前に，DMA 転送で気をつけなければならない制約について紹介しておきます．MFC の提供する DMA 転送は，どんなアドレスから始まるどんなサイズのデータでも転送できるというわけではありません．転送サイズとアドレスの 2 点において，厳しい制約があります．

まず，転送サイズ (バイト数) についてですが，MFC の DMA 転送には，**1，2，4，8 および 16 から 16,384 までの 16 の倍数のいずれかのバイト数の転送しか扱うことができない**という制約があります．次に，転送元と転送先のアドレスについてですが，MFC の DMA 転送には，**転送サイズが 16Byte 以上の場合はどちらのアドレスも 16 の倍数でなければならず，転送サイズが 16Byte 未満の場合は転送元のアドレスを 16 で割った余りと転送先のアドレスを 16 で割った余りが等しく，なおかつその余りが転送サイズの倍数でなければならない**という制約があります．たとえば 4Byte のデータを DMA 転送する場合，転送元のアドレスと転送先のアドレスの両方が 16 で割り切れるか，あるいは両者を 16 で割った際の余りがともに 4，8，12 のいずれかにならなければなりません (一方の余りが 4 で，もう一方の余りが 8 などは不可)．

MFC に対して，これらに違反したサイズやアドレスを含む DMA 転送の要求を発行することも可能ですが，その場合，MFC が違反のある DMA 転送の要求を処理し始めたところでエラーが発生し，SPE プログラムが終了してしまいます．

10.1.3 DMA 転送要求の仕方

DMA 転送は，MFC に対して DMA 転送の要求をコマンドの形で発行することで行います (MFC コマンド) *3．MFC へのコマンドは，チャネルを介して発行します．DMA 転送要求の発行で用いるチャネルの一覧を表 10.1 に示します．

表 10.1 MFC に対する DMA 転送要求の発行で用いるチャネルの一覧

チャネル名	R/W	深さ	ブロック	用途
`MFC_LSA`	W	1	しない	LS 側のアドレス指定
`MFC_EAH`	W	1	しない	EA 空間側のアドレスの上位 32bit 指定
`MFC_EAL`	W	1	しない	EA 空間側のアドレスの下位 32bit 指定
`MFC_Size`	W	1	しない	転送サイズ指定
`MFC_TagID`	W	1	しない	コマンドのタグ指定
`MFC_Cmd`	W	16	する	コマンド (転送方向情報などを含む) の指定

MFC に対するコマンドの発行で用いるチャネルはすべて書き込み専用です．`MFC_Cmd` チャネル以外のチャネルは，ノンブロッキング型のチャネルであるため，いくらでも値を書き換えることができます．`MFC_Cmd` チャネル以外の各チャネルに必要な情報を書き込んだ上で，`MFC_Cmd` チャネルにコマンドを書き込むと，他の 5 つのチャネルに書き込まれている値が回収され，`MFC_Cmd` チャネルに書かれたコマンドとセットになって MFC のキューに投入されます．`MFC_Cmd` チャネルのチャネルカウンタの値は，キューにコマンドが投入されると 1 減り，コマンドの処理が完了すると再び 1 増えます．

`MFC_LSA` チャネルなどに書いた値は，一度 `MFC_Cmd` チャネルにコマンドを書くと消えてしまいます．そのため，次に発行する DMA 転送の要求において，直前に発行した DMA 転送の要求と同じアドレスやタグを用いる場合でも，再度同じ値を書き込む必要があります*4．

`MFC_Cmd` チャネルの深さからも想像がつくかもしれませんが，MFC のコマンドキューの深さは 16 段となっており，最大で 16 個までの未処理のコマンドを蓄えることができます*5．16 個のコマンドが蓄積された状態で `MFC_Cmd` チャネルにさらにコマンドを書き込もうとすると，SPU はストールします．ストールを回避したい場合は，事前にチャネルカウンタを読んで，チャネルカウンタの値が 0 でないことを確認しておく必要があります．また，`MFC_Cmd` チャネルのチャネルカウンタの値は，MFC のコマンドキューの使用状況と連動するため，`MFC_Cmd` チャネルのチャネルカウンタの値を見ることで，MFC の処理状況を大雑把につかむことができます．

なお，`MFC_Cmd` チャネル以外の 5 つのチャネルについては，発行するコマンドの種類によっては表 10.1 に示したものとは異なる用途で用いられる場合があります．これについては別途紹介します．

---

*3 こう書くと MFC は DMA 転送要求以外の様々なコマンドにも対応しているように思われそうですが，実際にはほとんどのコマンドが DMA 転送を伴います．本書ではコマンドと DMA 転送要求はほぼ同義で使われていると考えてしまって差し支えありません．

*4 ただし `MFC_EAH` だけは例外で，ここに何も値を書かずにコマンドを `MFC_Cmd` に書いた場合は `MFC_EAH` には 0 が書かれているものとして処理されます．

*5 この 16 個という数字は Cell/B.E. に限った話らしく，異なる CBEA 準拠のプロセッサでは数が 16 であるとは限らないようです．

まずは最も基本的な 2 つの DMA 転送のコマンドを表 10.2 に示します．これらコマンドを MFC_Cmd チャネルに書き込むことで，MFC に対して DMA 転送が要求されます．

表 10.2　通常の DMA 転送を要求するコマンド

コマンド名	機能
MFC_PUT_CMD	LS から EA 空間への DMA 転送を要求
MFC_GET_CMD	EA 空間から LS への DMA 転送を要求

表 10.2 で示したコマンド名は，spu_mfcio.h というヘッダファイルで定義されています．DMA 転送を利用する場合は，SPE プログラム上でこのファイルを include しておく必要があります．

さて，ここで大事なことを 1 つお伝えしておきましょう．ここで紹介した 6 個のチャネルへの書き込みを行うことで MFC に DMA 転送要求を発行することができるというのはこれまで述べた通りなのですが，実は spu-gcc にはこれら 6 個のチャネルへの書き込みを一括で行うための複合組み込み関数が 2 つ用意されています．これらの複合組み込み関数を利用すれば，プログラム中の DMA 転送要求部分にいくつもの spu_writech 関数を書き並べる必要がなくなります．以下でこれらを紹介します．

spu_mfcdma64 : DMA 転送要求を発行する (64bit 版)

```
spu_mfcdma64(ls, eah, eal, size, tag, cmd);
```

spu_mfcdma64 は，MFC に対して DMA 転送要求を行うのに必要な 6 つのチャネルへの書き込みに変換される複合組み込み関数です．上記のフォーマットは，コンパイル時にリスト 10.1 のように変換されます．

リスト 10.1　spu_mfcdma64 の実際の処理

```
spu_writech(MFC_LSA, ls);
spu_writech(MFC_EAH, eah);
spu_writech(MFC_EAL, eal);
spu_writech(MFC_Size, size);
spu_writech(MFC_TagID, tag);
spu_writech(MFC_Cmd, cmd);
```

したがって，上記のフォーマットにおける spu_mfcdma64 の引数は，ls が LS のアドレス，eah が EA の上位 32bit，eal が EA の下位 32bit，size が転送サイズ，tag がコマンドにつけるタグ，そして cmd が DMA 転送要求のコマンドとなります．

spu_mfcdma32 : DMA 転送要求を発行する (32bit 版)

```
spu_mfcdma32(ls, eal, size, tag, cmd);
```

spu_mfcdma32 は，PPE 側の実行ファイルが 32bit バイナリの際に，MFC に対して DMA 転送要求を行う上で必要となる 5 つのチャネルへの書き込みに変換される複合組み込み関数です．上記のフォーマットの場合，コンパイル時にリスト 10.2 のように 5 つのチャネルへの書き込みに変換されます．

リスト 10.2 spu_mfcdma32 の実際の処理

```
spu_writech(MFC_LSA, ls);
spu_writech(MFC_EAL, eal);
spu_writech(MFC_Size, size);
spu_writech(MFC_TagID, tag);
spu_writech(MFC_Cmd, cmd);
```

PPE 側の実行ファイルが 32bit バイナリの場合，EA の上位 32bit は 0 になるので，設定が不要となる MFC_EAH チャネルへの書き込みが省略されています．PPE 側の実行ファイルを 32bit バイナリにして spu_mfcdma32 を使うようにすれば，DMA 転送の発行までに必要となる命令数を 1 つ減らすことができるので，もしかしたらプログラムの実行性能の向上にわずかながら効果があるかもしれません．また，PPE 側の実行ファイルが 32bit バイナリのときでも，第 2 引数に 0 を指定することで，spu_mfcdma64 を用いて DMA 転送を要求することができます．

10.1.4 DMA 転送の完了確認の仕方

次に，DMA 転送要求の処理が完了したことを確認する方法を紹介します．通常，過去に MFC に対して発行した DMA 転送要求の処理が完了したかどうかは，タグを使って追跡します．また，追跡の指示やタグの指定などは，チャネルを通じて行います．MFC に対する DMA 転送の完了の確認で用いるチャネルの一覧を表 10.3 に示します．

表 10.3 MFC に対する DMA 転送の完了確認で用いるチャネルの一覧

チャネル名	R/W	深さ	ブロック	用途
MFC_WrTagMask	W	1	しない	タグマスクの設定
MFC_RdTagMask	R	1	しない	現在設定されているタグマスクの取得
MFC_WrTagUpdate	W	1	する	DMA 転送のタグごとの完了状況調査依頼
MFC_RdTagStat	R	1	する	DMA 転送のタグごとの完了状況の取得

DMA 転送要求の発行に比べてだいぶわかりにくいのですが，10.1.1 でも述べたように，DMA 転送要求の完了状況を確認するには，事前に MFC に対してどのタグの DMA 転送要求の完了を確認するかを通知しておく必要があります[*6]．この “どのタグの” という部分を指定するのに用いるのが，MFC_WrTagMask チャネルへ書き込むことで指定する**タグマスク**になります．先ほど，タグには 0 から 31 までの番号を使えるということを述べましたが，タグマスクはこのタグの番号をビットパターンに置き換えたものになります．タグマスクは 32bit 幅で，LSB がタグ 0 に対応し，MSB がタグ 31 に対応します (**ビット番号と逆**である点に注意)．タグマスクを設定しておくと，MFC は後で DMA 転送の完了状況の調査依頼を受けた際に，タグマスクで 1 が立っているビットに対応するタグのついた DMA 転送についてのみ結果を返します．たとえば MFC_WrTagMask チャネルを介して設定されたタグマスクが 0x8000fffe であった場合，タグ 1 からタグ 15 までとタグ 31 の合計 16 種類のタグのついた DMA 転送に関して転送が完了しているかどうかを調べます．あるいはタグマスクとして 0xffffffff を指定すれば，すべての DMA

[*6] 厳密には “どのタググループに属した DMA 転送要求の完了を調査するのか” とすべきかもしれません．DMA 転送の完了はタグ単位で行うため，同じタグのついた (同じタググループに属した) DMA 転送要求については，個別に完了を調査することはできません．

転送の完了結果を調べることができます．タグマスクは，タグの問い合わせをする前ならいつどのタイミングでも更新することができます (さすがに問い合わせ中に書き換えるのは予測不可能な結果を招きそうで危険)．また，MFC_RdTagMask チャネルを読むことで，現在 MFC に設定されているタグマスクを確認することができます．

このタグマスクの情報を元に DMA 転送の完了状況の調査を依頼するためのチャネルが MFC_WrTagUpdate チャネルになります．このチャネルに対して書き込みを行うと，DMA 転送の完了状況の調査が始まります．そして，調査が完了すると，MFC_RdTagStat チャネルに DMA 転送の完了状況が格納されて，読み出せるようになります．

MFC_WrTagUpdate チャネルにどのような値を書くかによって，いつどのタイミングで調査結果を返すべきかを指定することができます．表 10.4 に，MFC_WrTagUpdate で指定可能な，調査結果を返すタイミングの一覧を示します．

表 10.4　MFC_WrTagUpdate チャネルへ書き込み可能な値と調査結果を返すタイミング

値	タイミング
MFC_TAG_UPDATE_ALL	タグマスクで指定したタグに対応するすべてのタググループの全 DMA 転送が完了した時点
MFC_TAG_UPDATE_ANY	タグマスクで指定したタグのどれか 1 つに対応するタググループの全 DMA 転送が完了した時点
MFC_TAG_UPDATE_IMMEDIATE	何も待たずに今の時点の完了状況を返す

たとえば MFC_WrTagUpdate チャネルに MFC_TAG_UPDATE_ALL を書いたとすると，その MFC に発行された，タグマスクで指定したタグのついたすべての DMA 転送が完了するのを待った上で，完了状況を MFC_RdTagStat チャネルから読み出せるようにセットします．タグマスクで指定したタグのついた DMA 転送がすべて完了する前に MFC_RdTagStat チャネルから値を読み出そうとすると，SPU はストールします．

また，これまで “完了した時点” と書いてきましたが，タグマスクで指定したタグのついた DMA 転送が，MFC_WrTagUpdate チャネルに書き込んだ時点で既にすべて完了していたり，あるいは過去に一度も MFC に対して発行したことがなかったりした場合でも，DMA 転送は完了したものと見なされ，MFC_RdTagStat チャネルから完了状況が読み出せるようになります[*7]．なお，MFC_WrTagUpdate チャネルはブロッキング型のチャネルであるため，連続して値を書こうとすると SPU がストールしますが，一度値を書いてからしばらくすると，調査対象の DMA 転送が完了する前でも再び書き込み可能となります．

MFC_RdTagStat チャネルから読み出される値は，タグマスクで 1 を立てたビットから MFC のキューにまだ残っている DMA 転送要求のタグのビットを下げたものになります (要求がすべて完了しているタグに対応したビットが立ったまま残る)．よって，MFC_TAG_UPDATE_ALL を MFC_WrTagUpdate チャネルに書いた場合，後で MFC_RdTagStat チャネルから読み出せる値はタグマスクとまったく同じものになるはずです．

[*7] 過去に一度も発行したことのないタグでも “完了した” という扱いとなることから，MFC_RdTagStat チャネルから読み出せる情報は “どのタグの DMA 転送が完了したのか” という情報であると考えるよりも，“どれがまだ残っているか” という情報であると考えるのが自然なのかもしれません．

ところで，ここで紹介した MFC_WrTagUpdate チャネルへの書き込みと MFC_RdTagStat チャネルからの読み出しについても，これを一括で行う複合組み込み関数が spu-gcc に用意されています．以下で紹介します．

spu_mfcstat：DMA 転送要求の完了状況取得

```
d = spu_mfcstat(mode);
```

spu_mfcstat は，前述の MFC_WrTagUpdate チャネルへの書き込みと，MFC_RdTagStat チャネルからの読み出しに変換される複合組み込み関数です．上記のフォーマットの場合，コンパイル時にリスト 10.3 に示す処理に変換されます．引数である mode は，内部で MFC_WrTagUpdate へ書き込まれることから，表 10.4 に示した値のいずれかを設定することになります．また，d には MFC_RdTagStat チャネルから読み出した DMA 転送の完了状況が格納されます．

リスト 10.3 spu_mfcstat の実際の処理

```
spu_writech(MFC_WrTagUpdate, mode);
d = spu_readch(MFC_RdTagStat);
```

10.1.5 DMA 転送対象領域のアドレスの調整

DMA 転送を行う場合，まず，変数を DMA 転送可能なアドレスやサイズで確保しなければなりません．サイズは型や配列で簡単に調整できますが，アドレスに関しては少々厄介です．

自動変数やグローバル変数などのアドレスを調整する場合は，リスト 10.4 のように変数宣言の際に GCC のディレクティブを与えて指示するのが一番簡単です．リスト 10.4 の例で，配列の宣言の後ろについている “__attribute__((aligned(16)))” がディレクティブになります．括弧の中の数字を用いて，この配列の先頭が 16 の倍数のアドレスになる (16Byte 境界にアラインする) ように確保するということを指示しています．このディレクティブは spu-gcc や ppu-gcc 固有のものというわけではなく，他のアーキテクチャ向けの GCC でも普通に利用できます．

リスト 10.4 自動変数を 16Byte 境界にアラインした状態で確保する際の記述例

```
    unsigned dma_buf[4] __attribute__((aligned(16)));
```

もう少しこのディレクティブについて詳しく紹介します．このディレクティブをつけた変数や配列がどのようなアドレスから始まるかについては，__attribute__((aligned(n))) の n を変更することで自由に指定することができます．ただし，n には 2 のべき乗の数しか指定することができません．また，自動変数の場合，n に 16 より大きい数 (32 や 64 など) を指定しても，その数にアラインするとは限りません (少なくとも ppu-gcc や spu-gcc では 32 や 64 を指定しても 16 の倍数になることしか保証されない) *8．一方，グローバル変数であれば，__attribute__((aligned(n))) を用いて 32Byte アラインや 64Byte アラインなどを行うことが可能です．__attribute__((aligned(n))) についてのより詳細な情報は GCC の info に

*8 どうしても自動変数でやりたい場合は，ちょっと大きめに配列を確保して，後で開始アドレスをずらすなどの方法を取る必要があります．

書かれていますので，もっと詳しく知りたい場合はそちらを参照してください．

また，アラインされた変数や配列を動的に確保するには，PPE の場合，`malloc` の代わりに `posix_memalign` 関数を用います．一方，SPE の場合は，`malloc.h` を `include` しておくことで `memalign` 関数を使うことができます．

10.1.6 DMA 転送の例

ここまでの説明でようやく DMA 転送のコードを紹介する準備が整いました．DMA 転送で EA 空間から LS に値を読み出す際の，SPE 側のコードの例をリスト 10.5 に示します．

リスト 10.5 DMA 転送でデータを EA 空間から LS に読み出すコードの例

```
#include <spu_mfcio.h>
...
#define TAGID 3
...
    // DMA用のバッファを16Byte境界にアラインした状態で確保
    unsigned dma_buf[4] __attribute__((aligned(16)));
    ...
    // タグマスクを設定 (TAGIDのみ完了確認するように)
    spu_writech(MFC_WrTagMask, 1 << TAGID);

    // 16Byteのデータを ((eah << 32) | eal) からdma_bufにDMA転送要求
    spu_mfcdma64(dma_buf, eah, eal, 16, TAGID, MFC_GET_CMD);
    // タグマスクで指定したタグの全要求が完了となるまで待機
    spu_mfcstat(MFC_TAG_UPDATE_ALL);
```

リスト 10.5 では，まずタグマスクを設定し，次に DMA 転送要求を発行して，最後に DMA 転送の完了を待つということを行っています．DMA 転送要求の発行に対応する `spu_mfcdma64` の呼び出しにおいて，引数として与えられている `eah` や `eal` は，SPE が好き勝手に決める値ではなく，PPE 側から教えてもらうべき値です．リスト 10.5 では省略していますが，これらは事前にメールボックスやシグナル，`main` の引数などの何らかの手段で PPE から受け取っておく必要があります．

また，リスト 10.5 のコードでは，DMA 転送の要求をする前にタグマスクを設定していますが，タグマスクは `spu_mfcstat` を呼ぶ前に設定されていればよいので，`spu_mfcdma64` の後に設定しても構いません．ここでは，“タグマスクは `spu_mfcstat` を呼ぶ直前に書くべきものである” という誤解を避けるために，あえて前の方に書いています．

`spu_mfcstat` に着目すると，この複合組み込み関数は値を返すにも関わらず，プログラム中で戻り値を受け取っていません．これは，`spu_mfcstat` の引数に `MFC_TAG_UPDATE_ALL` を指定していることから，戻り値としてタグマスクに指定したビットマスクと同じ値が返ってくることが明らかであるためです．なお，蛇足ではありますが，ここではタグマスクでタグを 1 個しか指定していないことから，`MFC_TAG_UPDATE_ANY` を用いたとしても同じ結果となります．

ところで，リスト 10.5 の例では，`spu_mfcdma64` で DMA 転送を要求した直後に `spu_mfcstat` で DMA 転送の完了を確認していますが，本来このような使い方は好ましくありません．DMA 転送の完了をただ何もせずに待つのは非常に無駄なので，通常はできるだけ早めに DMA 転送の指示を出し，その後に DMA 転送の完了を必要としない別の処理を進め，DMA 転送で読み出したデータを使う (あるいは DMA 転送での書き出しを指示したデータを書き潰す) ギリギリ直前に `spu_mfcstat` を呼んで DMA 転送の完了を待つのが理想的です．要求の発

行から完了待ちを行うまでの間に十分な時間の開きがあるならば，完了待ちは実際に “待ち” とならないはずです．この辺りに関しては，10.7.1 でもう少し詳しく解説します．

10.1.7 DMA 転送のサイズとアドレスによる性能差の実際

文献 [5] などによると，DMA 転送で主記憶との間で転送を行う場合，主記憶と LS の両方の転送アドレスが 128Byte 境界にアラインされており，なおかつ転送サイズが 128 の倍数のときが最も効率がよいとあります．これは PPE のキャッシュのラインサイズと，DMA 転送で LS へアクセスする際のデータ幅がともに 128Byte であることに起因するものと思われます．実際に 128Byte 境界にアラインした主記憶と LS との間でデータ転送を行い，データサイズごとに DMA 転送要求を発行してから完了が確認できるまでの時間をデクリメンタで測定してみると，128Byte ごとに転送時間が増えていくことが確認できます．測定した結果を図 10.1 に示します．

また，文献 [1] などでは，16Byte 未満の DMA 転送については性能が低く，非常に効率が悪いため極力避けるべきとされています．しかしながら，こちらは実際に測ってみると，図 10.2 に示すように，主記憶への DMA 書き込みの方は 16Byte 転送時に比べてレイテンシが増大するものの，主記憶からの DMA 読み出しの方は 16Byte 転送時とレイテンシの差がほとんどないことがわかります．“性能が低い” という曖昧な表現に惑わされがちですが，16Byte 未満であっても，少なくとも単発の主記憶からの DMA 読み出しに関しては，16Byte や 128Byte のデータを転送した際のレイテンシと変わらないので，先頭の 8Byte があればよいところを無理して 128Byte 単位で主記憶から読み出すようなことをする必要はないでしょう．

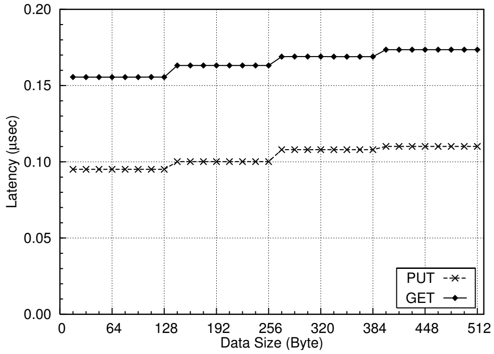

図 10.1　16Byte から 512Byte までの DMA 転送のレイテンシ

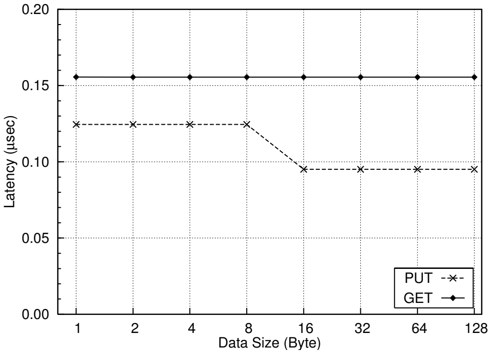

図 10.2　16Byte 未満の DMA 転送のレイテンシ

一方，主記憶側と LS 側のどちらかで 128Byte 境界をまたぐような DMA 転送を行った場合，レイテンシが増大します．これは，境界をまたぐと，またがなかった場合に比べて LS や主記憶へのアクセス回数が 1 回分増えてしまうためです．128Byte 境界にアラインした LS と主記憶の領域間で，0 から 240 までの任意の 16 の倍数のオフセットを設定して 32Byte の DMA 転送を行った際の，主記憶への DMA 書き込みのレイテンシを図 10.3 に，主記憶からの DMA 読み出しのレイテンシを図 10.4 に，それぞれ示します．転送サイズが 32Byte なので，LS のアドレスや EA のオフセットが 112 や 240 のときに 128Byte 境界をまたぐことになります．

図 10.3 および図 10.4 より，LS 側で 128Byte 境界をまたいだ場合の方が，主記憶側で 128Byte

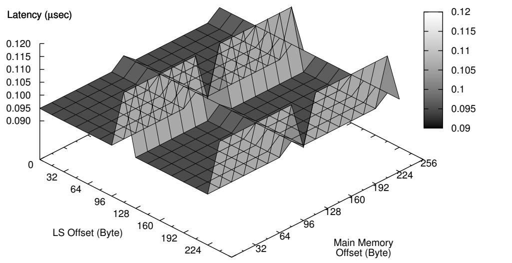

図 10.3　32Byte の主記憶への DMA 書き込みのレイテンシ

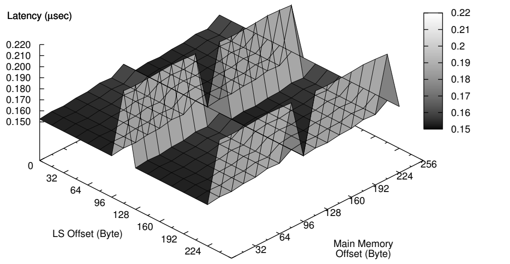

図 10.4　32Byte の主記憶からの DMA 読み出しのレイテンシ

境界をまたいだ場合に比べてレイテンシが大きく増えていることがわかります．おそらく DMA 転送による LS への 128Byte 幅での連続アクセスが 16 サイクルに 1 回*9しか行えないためでしょう．また，境界をまたぐ場合でも，主記憶と LS の 128Byte 境界からのオフセットが等しい (アラインメントが一致している) 場合の方が，そうでない場合に比べてレイテンシが低くなります．これは，オフセットが等しい場合には，データを転送途中で格納先に合わせて並べ替える必要がなくなるためだと考えられます．これらの結果から，32Byte から 112Byte までのサイズのデータを転送する際は，とにかく LS 側で 128Byte 境界をまたぐのを最優先で避け，できれば主記憶側でも 128Byte 境界をまたがないようにし，どうしてもまたいでしまう場合は両者のアドレスを 128 で割った余りが揃った転送とすべきであるということが言えそうです．

では，絶対に 128Byte 境界をまたぐことのない 16Byte の転送はどうなのでしょうか．測ってみると，興味深い結果が得られます．128Byte 境界にアラインした LS と主記憶の領域間で，0 から 112 までの任意の 16 の倍数のオフセットを設定して 16Byte の DMA 転送を行った際のレイテンシを図 10.5 と図 10.6 に示します．

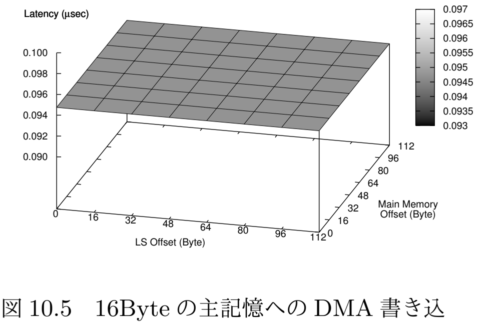

図 10.5　16Byte の主記憶への DMA 書き込みのレイテンシ

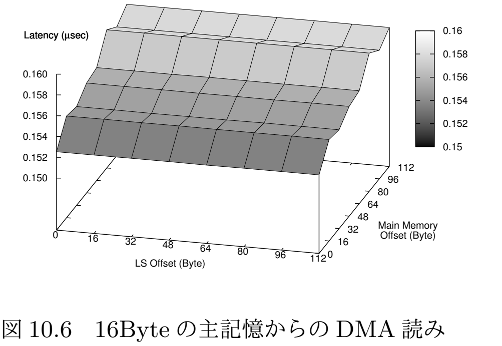

図 10.6　16Byte の主記憶からの DMA 読み出しのレイテンシ

*9 厳密には 8 サイクルに 1 回なのですが，片方は読み出し専用のサイクル，もう片方は書き込み専用のサイクルとなるため，読み出し (もしくは書き込み) ばかり行ってる場合は 16 サイクルに 1 回のアクセスとなります．

LS 側の転送アドレスについては，128Byte のどの 16Byte を転送対象としても，レイテンシにほとんど影響が出ていません．一方，主記憶側の転送アドレスは，主記憶への DMA 書き込みでは，どの 16Byte を対象としてもレイテンシに変化がありませんが，主記憶からの DMA 読み出しでは，他と違う傾向を見せます．主記憶からの DMA 読み出しの場合，主記憶側のアドレスを 128Byte 境界間の前半にしたときよりも，後半にしたときの方が明らかにレイテンシが大きくなっています．

次に，オフセットの範囲を 0 から 2,032 にまで広げて同様の測定を行った結果を，図 10.7 と図 10.8 に示します．図 10.7 より，主記憶への DMA 書き込みのレイテンシは，128Byte 境界内であれば主記憶と LS のどちらのアドレスのオフセットにも影響されることはないものの，主記憶側のアドレスが 128Byte 単位でずれると大きく変化することがわかります．また，図 10.8 からは，図 10.6 で見られた主記憶のオフセットの増加によるレイテンシの増加が，128Byte 周期で繰り返し発生していることがわかります．

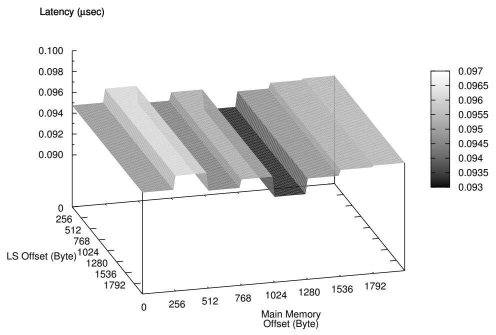

図 10.7　2,048Byte 範囲での 16Byte の主記憶への DMA 書き込みのレイテンシ

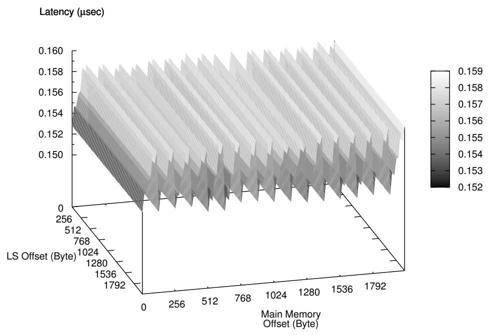

図 10.8　2,048Byte 範囲での 16Byte の主記憶からの DMA 読み出しのレイテンシ

これらの結果から何らかの結論を導くのはなかなか難しいところではありますが，16Byte の主記憶からの DMA 読み出しを行う場合は，とにかく主記憶側のアドレスをなるべく 128Byte 境界間の前半に確保した方が性能面で有利であるということは言えそうです．

最後に，図 10.8 などと同様に測定した LS 間での 16Byte の DMA 転送のレイテンシを図 10.9 および図 10.10 に示します．全体的に，主記憶との間の転送に比べて低レイテンシで転送できることがわかります．また，主記憶と LS の間での転送で見られた，転送相手のオフセットによる規則的なレイテンシの変化などは見られません．この結果から，LS 間の DMA 転送に関しては，アドレスのアラインメントに関して最低条件を満たしておけば，それ以上神経質になる必要はなさそうだということが言えそうです．

10.2　フェンスとバリア

これまで述べたように，MFC に発行された DMA 転送要求などのコマンドは，基本的に MFC 内でキューに格納されて処理されます．これによって，SPU からは最大で 16 個までの DMA 転送要求を，先に発行した DMA 転送要求が処理されたかどうかを確認することなく連続して発行

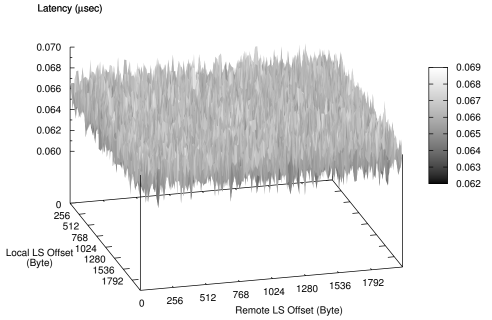

図 10.9 2,048Byte 範囲での 16Byte の LS への DMA 書き込みのレイテンシ

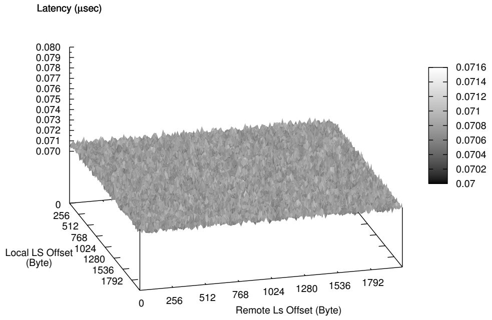

図 10.10 2,048Byte 範囲での 16Byte の LS からの DMA 読み出しのレイテンシ

することができます．が，ここで 1 つ注意すべき点があります．MFC のキューは FIFO ではありません．したがって，キューの中の DMA 転送要求は，**必ずしも SPU が発行した順に処理されるとは限らないのです．**

図 10.11 に MFC のキューのイメージを示します．MFC に発行された DMA 転送要求 (図の中の円) は右から左へ発行順に並んでいますが，MFC がその中のどの DMA 転送要求を処理するかは，その場の状況に応じて変化します．したがって，図に示すように，DMA 転送要求をキューに投入した順番と，実際に発生する DMA 転送の順番が一致しなくなる可能性があります．

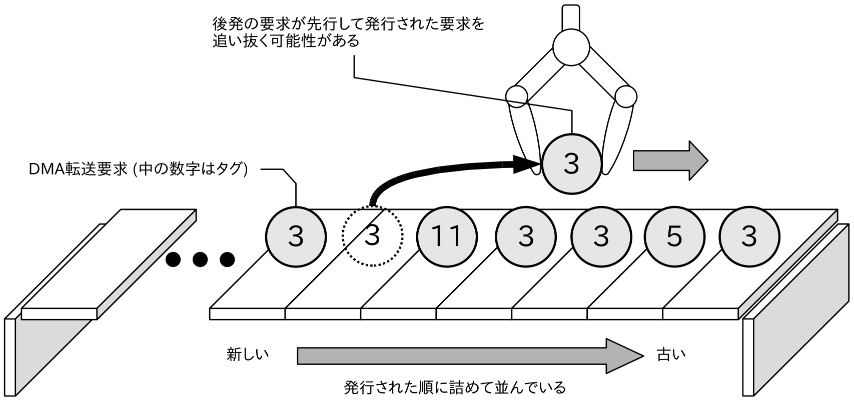

図 10.11 MFC のキューのイメージ

SPE のプログラムで，毎回 DMA 転送を 1 回発行するたびに完了を待つような実装としている場合や，どの順番で DMA 転送が行われたとしても最終的にすべての DMA 転送が処理されればそれでよいという状況であれば，これが問題となることはないでしょう．しかしながら，先にデータを DMA 転送で主記憶に書き込んだ上で，最後に書き込みが完了したことを示すフラグを主記憶へ書き込むといった，転送の順番が重要となる転送処理では，このような MFC による DMA 転送要求の追い越しに備えて，フラグの書き込みの DMA 転送を要求する前にデータの書き込みの DMA 転送の完了を待つ処理を挟む必要が生じてしまいます．ここで紹介するフェンスやバリアは，このような状況において，DMA 転送の完了を待つ処理を挟まずに済ませるための

仕組みになります．

先にそれぞれの意味を説明します．フェンスもバリアも通常の DMA 転送要求に付随する属性です．フェンス属性のついた DMA 転送要求を MFC に発行すると，**それ以前にキューに投入された同じタグのついた DMA 転送要求よりも必ず後に処理される**ようになります．すなわち，フェンス属性をつけた場合，その DMA 転送要求に対して“先に発行している同じタグの DMA 転送要求を追い抜かない”という指示を与えることになります．ただし，フェンス属性のついた DMA 転送要求よりも後に MFC に発行した同じタグの DMA 転送要求は，フェンス属性のついた DMA 転送要求を追い抜いて処理される可能性があります．

フェンス属性のついた DMA 転送要求が，他の DMA 転送要求に対してどのように作用するのかについてまとめたものを図 10.12 に示します．図の“フェンス”と書かれた円が，フェンス属性のついた DMA 転送要求となります．この例では，フェンス属性はタグが 3 の DMA 転送要求につけられているため，この DMA 転送要求は，それより先行して発行されている同じタグ (タグが 3) の DMA 転送要求よりも必ず後に処理されることが保証されます．一方，フェンス属性のついた DMA 転送要求よりも先行して発行されている，異なるタグのついた DMA 転送要求に対する影響はなく，この先行する DMA 転送要求とフェンス属性のついた DMA 転送要求のどちらが先に処理されるかは不確定です．また，フェンス属性のついた DMA 転送要求よりも後に発行された DMA 転送要求については，タグによらずフェンス属性の影響を受けることはありません．そのため，これら後発の DMA 転送要求についても，フェンス属性のついた DMA 転送要求とどちらが先に処理されるかは不確定です．

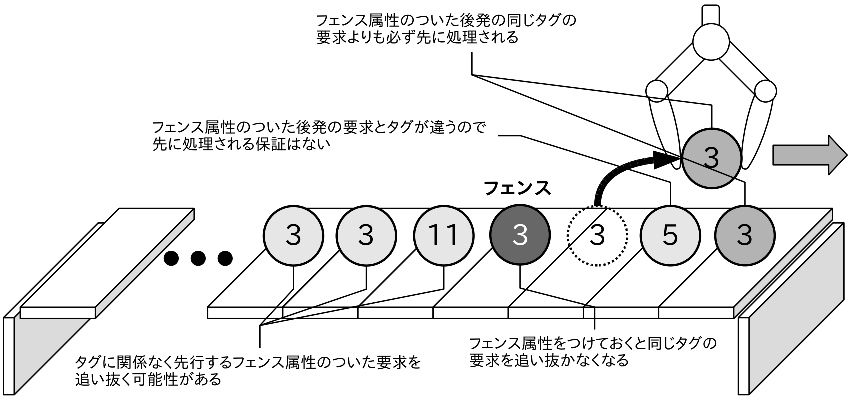

図 10.12　フェンス属性のついた DMA 転送要求

一方，バリア属性のついた DMA 転送要求を MFC に発行すると，フェンスと同様に，**それ以前にキューに投入された同じタグのついた DMA 転送要求よりも必ず後に処理される**ことが保証され，さらに，**それより後に発行される同じタグの DMA 転送要求よりも必ず先に処理される**ことも保証されます．すなわち“先に発行している同じタグの DMA 転送要求を追い抜かず，さらに後から発行する同じタグの DMA 転送要求に追い抜かれない”という属性を与えることができます．

バリア属性のついた DMA 転送要求が，他の DMA 転送要求に対してどのように作用するのかについてまとめたものを図 10.13 に示します．図の“バリア”と書かれた円が，バリア属性の

ついた DMA 転送要求となります．この例では，バリア属性はタグが 3 の DMA 転送要求につけられているため，この DMA 転送要求は，フェンスと同じように，それより先行して発行されている同じタグ (タグが 3) の DMA 転送要求よりも必ず後に処理されることが保証されます．また，それに加え，この DMA 転送要求が，それより後に発行された同じタグ (タグが 3) の DMA 転送要求よりも必ず先に処理されることも同時に保証されます．

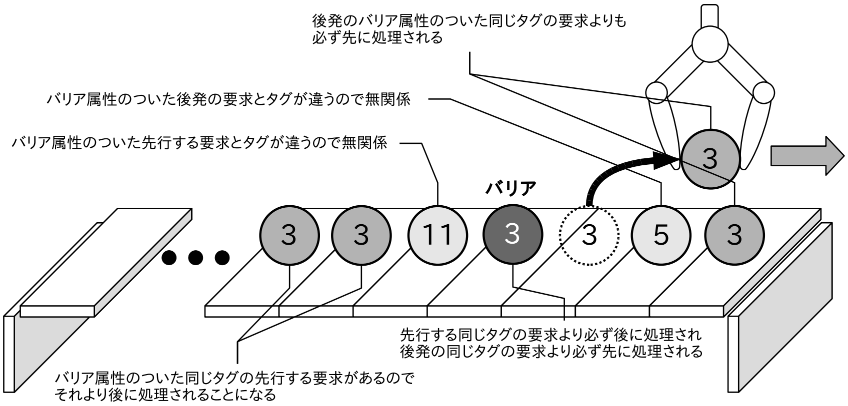

図 10.13 バリア属性のついた DMA 転送要求

DMA 転送要求にフェンスやバリアの属性をつけるには，MFC_Cmd チャネルに書き込むコマンド自体を変更します．表 10.5 に，フェンスやバリアの属性を伴う DMA 転送要求のコマンドの一覧を示します．

表 10.5 フェンスやバリアの属性のついた DMA 転送を要求するためのコマンド

コマンド名	機能
MFC_PUTF_CMD	フェンス属性のついた MFC_PUT_CMD
MFC_GETF_CMD	フェンス属性のついた MFC_GET_CMD
MFC_PUTB_CMD	バリア属性のついた MFC_PUT_CMD
MFC_GETB_CMD	バリア属性のついた MFC_GET_CMD

通常の DMA 転送要求をフェンス属性やバリア属性のついたものに変更する場合は，spu_mfcdma64 や spu_mfcdma32 の最後の引数を表 10.5 に示したものに置き換えます．

なお，フェンスやバリアで追い越しなどの禁止を指示できるのは，あくまで同じタグのついた (同じタググループに属した) DMA 転送要求の間に限定されます．どうしても異なるタグのついた DMA 転送要求の間で追い越しなどを禁止したい場合は，コマンドに MFC_BARRIER_CMD を用います．MFC_BARRIER_CMD は，DMA 転送要求ではなく，それより前にキューに投入された DMA 転送要求と，それより後にキューに投入される DMA 転送要求の間で処理の順序の入れ替えが発生しないことを保証する特殊な MFC コマンドになります．DMA 転送要求ではありませんが，MFC_Cmd チャネルにコマンドを書き込むことで発行するので，spu_mfcdma64 や spu_mfcdma32 で発行することができます[*10]．なお，このコマンドにもタグを指定することが

[*10] その場合一応タグとコマンド以外の引数は 0 にしておくのがよさそうです．

できますが，ここで指定したタグがバリアの対象に影響することはありません*11.

10.3 リスト DMA

これまで紹介してきた DMA 転送は，いわば通常の DMA 転送で，フェンスやバリアといった属性がついたとしても，とにかく DMA 転送要求 1 個につきデータの転送を 1 回だけ行うというものでした．これに対し，リスト DMA は，1 個の DMA 転送要求で複数回の DMA 転送を行う特殊な DMA 転送になります.

10.3.1 リスト DMA の概要

リスト DMA では，**LS 上の連続した領域と EA 空間上の複数の不連続な領域との間**での DMA 転送を行います．EA 空間上の複数の不連続な領域は，あらかじめ LS 上に確保しておいた DMA 転送のアドレスとサイズのリスト (C 言語的に言うと構造体の配列) で指定します．これを用いると，主記憶上で飛び飛びに配置されているデータを LS 上のひと続きの領域に集約したり (ギャザ)，あるいは逆に LS 上の連続した領域にあるデータを主記憶上に分散させたり (スキャッタ) するといった，通常の DMA 転送では DMA 転送要求を複数回発行しなければならないような処理を，1 回の DMA 転送要求でまとめて行うことが可能となります.

リスト DMA を行うには，まず `mfc_list_element_t` 構造体の配列を LS 上に確保します. `mfc_list_element_t` 構造体は，図 10.14 に示すように 64bit 幅で，`notify`，`size` および `eal` のメンバを持ちます．`eal` は EA 空間側の転送領域のアドレスの下位 32bit を指定するのに用い，`size` はそのアドレスとの間で DMA 転送するサイズを指定するのに用います．`notify` は，1 にした場合，その要素の転送が完了したところでリスト DMA の処理が一時中断 (ストール) するようになります．これについての詳細は後で述べます.

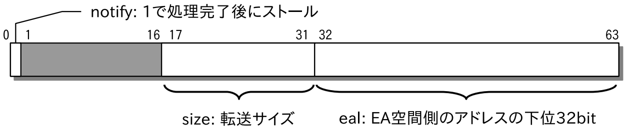

図 10.14 `mfc_list_element_t` の構造

このリストの先頭アドレスと，リストに含まれない情報 (EA の上位 32bit やタグなど) を MFC に対してコマンドの形で通知することで，リスト DMA による転送が始まります．なお，リストの要素に含まれない情報は，リストによって行われる全 DMA 転送で共通のパラメータとなります．たとえば転送方向は全 DMA 転送で共通のパラメータとなります．この制約により，単一のリスト DMA 転送で，LS から EA 空間への転送と EA 空間から LS への転送の両方を行うといったことはできないことになります.

リスト DMA の処理が開始すると，リストの先頭から 1 要素ずつ MFC に読み出され，リスト

*11 おそらくここでつけるタグは，後で `MFC_BARRIER_CMD` 自体の完了を `spu_mfcstat` 関数などで調べるのに用いるのではないかと思うのですが，筆者はそもそも `MFC_BARRIER_CMD` 自体を真面目に使ったことがないので何とも ….

の要素で指定された DMA 転送が行われます．リスト DMA では，同じリストで発行される個々の DMA 転送の間での追い越しは発生せず，リストの順番通りに DMA 転送が行われることが保証されています．リストの最後の要素の DMA 転送が完了すると，リスト DMA の処理が完了したという扱いとなります．リスト DMA の完了も，通常の DMA 転送と同じようにタグを用いて確認することができます．

10.3.2　リスト DMA の制約

リスト DMA においても，個々の転送におけるアラインメントやサイズの制約は，通常の DMA 転送と同様に存在します．リストの 1 要素で転送可能な最大サイズは，通常の DMA 転送と同じく 16,384Byte となります．また，リストの要素の数は最大 2,048 個までとなっており，リストは LS 上で 8Byte 境界にアラインして配置されている必要があります．

LS 側の転送対象領域のアドレスは，通常は各要素の処理が終わるたびにその要素で指定したサイズ分増えますが，要素に 16Byte 未満のサイズが指定されていた場合は，例外的にサイズ分ではなく，次の 16Byte 境界まで一気に増えてしまいます．したがって，EA 空間上に分散している 4Byte のデータを，うまくアラインメントを調整しながらいくつも集めて LS 上に密に配置するようなことは，リスト DMA ではできません (必ず 16Byte アラインになるのでスカスカになる)．

10.3.3　リスト DMA の要求の仕方

リスト DMA も，通常の DMA 転送と同様に，`spu_mfcdma64` や `spu_mfcdma32` を呼ぶ (あるいはチャネルに必要事項を直接書き込む) ことで MFC に要求を発行します．ただし，リスト DMA の発行の際は，これら関数への引数の与え方 (あるいはチャネルに直接書く情報) が通常の DMA 転送のときから一部変更になります．以下で，`spu_mfcdma64` の引数を例に違いを紹介します．他の場合については適宜読み替えてください．

まず，第 1 引数には，通常の DMA 転送の要求と同様に LS 側の転送対象領域の開始アドレスを指定します．ただし，ここで指定したアドレスは，リストの最初の要素の DMA 転送で用いるアドレスでしかなく，リストの要素が処理されるたびに LS 側のアドレスは各要素で指定したサイズ分増えていきます (前述の通り，16Byte 未満の転送の際には例外あり)．

第 2 引数には，転送相手となる EA の上位 32bit を指定します．この上位 32bit と各要素で指定した下位 32bit を組み合わせたアドレスが，各要素の転送対象の EA 側のアドレスになります．この仕様により，EA の上位 32bit は固定となるため，途中で EA の 4GByte 境界をまたぐようなリスト DMA はできないことになります[*12]．

第 3 引数には，通常の DMA 転送の要求であれば，転送対象となる EA の下位 32bit を渡しますが，リスト DMA の場合はここにリストの先頭アドレスを指定します．ここが一番不自然なので注意が必要です (本来 EA の下位 32bit を書くところに LS のアドレスを書くので)．また，第

[*12] 4GByte 境界のギリギリ直前から転送が始まるように下位 32bit を指定しておいて，その要素で指定したサイズ分転送している間に 4GByte 境界をまたぐことは可能らしいです．でも，それができることにどんな価値があるのかは筆者にはよくわかりません …．

4 引数についても，転送サイズではなくリストのサイズを指定します．このリストのサイズは，リストの要素数ではなく，**リスト全体のバイト数**になります．したがって，ここでは 8 (要素数 1 のとき) から 16,384 (要素数 2,048 のとき) までの，8 の倍数のいずれかを指定することになります．

第 5 引数には通常の DMA 転送と同じくタグを指定します．そして第 6 引数でリスト DMA を行うためのコマンドを指定します．リスト DMA を行うためのコマンドを表 10.6 に示します．

表 10.6　リスト DMA を要求するコマンド

コマンド名	機能
`MFC_PUTL_CMD`	LS から EA 空間へ転送するリスト DMA
`MFC_GETL_CMD`	EA 空間から LS へ転送するリスト DMA
`MFC_PUTLF_CMD`	フェンス属性のついた `MFC_PUTL_CMD`
`MFC_GETLF_CMD`	フェンス属性のついた `MFC_GETL_CMD`
`MFC_PUTLB_CMD`	バリア属性のついた `MFC_PUTL_CMD`
`MFC_GETLB_CMD`	バリア属性のついた `MFC_GETL_CMD`

表 10.6 に示したように，リスト DMA にもフェンスやバリアの属性をつけることができます．既に述べた通り，単一のリスト DMA の中で発生する DMA 転送の間では，追い越しは発生せず，リストにある順番通りに転送が行われますが，別のコマンドで発行された DMA 転送やリスト DMA との間では，通常の DMA 転送と同じく，コマンドを発行した順番で処理されることが保証されていません．このような場合，通常の DMA 転送と同様に，リスト DMA にフェンスやバリアの属性をつけることで，同じタグのついた他の DMA 転送要求やリスト DMA の要求との間での処理の順番をはっきりさせることができます．

10.3.4　リスト DMA の例

リスト DMA の使用例をリスト 10.6 に示します．この例では，`dma_buf` 上に連続配置されたデータを EA 空間の離散した領域に転送しています．まず，最初のループで `dma_buf` 上のデータを先頭から `BLOCK_SIZE`Byte ずつ，EA 空間上の `INTERVAL`Byte 間隔の各領域に転送するようにリストを初期化しています．次に，`spu_mfcdma64` でリスト DMA の要求を発行しており，コマンドとして `MFC_PUTL_CMD` を渡すことで，EA 空間側への書き込みを行うことを指示しています．リスト DMA の要求を発行してから完了を待つまでの部分に関しては，引数の使い方が異なる点を除いて，これまで紹介してきた通常の DMA 転送と何ら変わりがないことがわかります．

10.3.5　リスト DMA のストール

リスト DMA で，リストの要素の `notify` メンバを 1 に設定しておくと，(その要素がリストの最後の要素でなければ) その要素の転送が完了したところでリスト DMA の処理がストールします．ストールすると，`MFC_RdListStallStat` チャネルに，現在どのタグのリスト DMA がストールしているのかという情報がビットマスクの形で格納され，SPU からの読み出しが可能となります．ビットマスクの構造はタグマスクと同様で，タグ 0 のリスト DMA がストールした場

リスト 10.6　リスト DMA による DMA 書き込みを行うコードの例

```
#define LIST_NUM 128
#define BLOCK_SIZE 16
#define INTERVAL 128
#define TAGID 3

// LIST_NUM個のBLOCK_SIZEバイトのデータを連続して詰める領域を確保
unsigned char dma_buf[BLOCK_SIZE * LIST_NUM] __attribute__((aligned(16))) = {0};
mfc_list_element_t dma_list[LIST_NUM] __attribute__((aligned(16))) = {{0}};
...
    // リストDMAのリストを初期化
    for (i = 0; i < LIST_NUM; i++) {
        dma_list[i].size = BLOCK_SIZE;
        dma_list[i].eal = eal;
        eal += INTERVAL;
    }

    spu_writech(MFC_WrTagMask, 1 << TAGID);

    // リストDMAでdma_buf上の連続したデータをリストの要素の指す領域に書き込む
    spu_mfcdma64(dma_buf, eah, (unsigned)dma_list,
                 sizeof(mfc_list_element_t) * LIST_NUM, TAGID, MFC_PUTL_CMD);
    spu_mfcstat(MFC_TAG_UPDATE_ALL);
```

合に LSB に 1 が立ち，タグ 31 のリスト DMA がストールした場合は MSB に 1 が立ちます．なお，リスト DMA がストールしていない状態で `MFC_RdListStallStat` チャネルを読みに行くと，SPU の方がストールしてしまうので，注意してください．

ストールしたリスト DMA は，`MFC_WrListStallAck` チャネルに値を書き込むことでストールを解除することができます．その際，`MFC_WrListStallAck` チャネルに書き込む値は，タグのビットマスクではなくタグの値そのものになります．たとえばタグが 3 のリスト DMA のストールを解除したかったら，ここに 8 (`1 << 3`) ではなく 3 を書きます．したがって，このチャネルで一度にストールを解除できるのは，同じタグを持った (同一のタググループに属する) リスト DMA に限られます．なお，このチャネルはノンブロッキング型のチャネルなので，値を書き放題なのですが，ストールしていないリスト DMA のタグを書くとよからぬことが起こる場合があるようなので，余計なことはしない方がよいでしょう．

また，リスト DMA では，要素の先読みなどは行われないことから，リスト DMA がストールしている間に，今後処理されるであろうリストの要素を書き換えて，リストを修正することが可能です．たとえばリスト DMA がストールしている間に，今後処理される予定の要素の転送サイズを 0 に書き換えることで，その要素の転送をスキップさせることができます．

なお，リスト DMA の何個目でストールしているのかを知る方法が気になるかもしれませんが，おそらくこれは用意されていません．同じタグのついたストールする可能性のあるリスト DMA が複数同時に走っていたりでもしない限りは，タグからどのリストの DMA が止まったかを判断でき，そのリストを先頭から手繰れば何回中断する可能性があるのかを調べることができます[*13]．よって，“このタグのリスト DMA が中断したのが何回目か” ということをプログラム中で数えておけば，リスト上に `notify` の設定された要素が複数あっても，リストの何要素目を処理したところで中断したのかを正確に知ることができます．逆に考えると，ストールする可能性のある複数のリスト DMA に同じタグをつけて同時に走らせるようなことをすると，どちらが

[*13] もちろんプログラミングの段階で把握できているという場合もあるでしょう．

ストールしたのか見分けがつかなくなり，管理が破綻してしまいます．

ストールするリスト DMA の使い方の例をリスト 10.7 に示します．この例では，リスト 10.6 で示したリスト DMA をベースに，すべての要素がストールするように `notify` メンバに 1 を設定するよう変更を加えています．また，リスト DMA の要求を発行した後，(リストの要素数 − 1) 回のストールを待っています．(リストの要素数) 回ではなく (リストの要素数 − 1) 回だけ待つようにしているのは，リストの最後の DMA 転送については `notify` が 1 でもストールが発生しないためです．うっかり最後の要素の分も `MFC_RdListStallStat` チャネルを読みに行ってしまうと，SPU の方がストールしてしまうので注意が必要です．

リスト 10.7 ストールを伴うリスト DMA による DMA 書き込みを行うコードの例

```
for (i = 0; i < LIST_NUM; i++) {
    dma_list[i].size = BLOCK_SIZE;
    dma_list[i].eal = eal;
    // notifyメンバに1を書いておく
    dma_list[i].notify = 1;
    eal += INTERVAL;
}

spu_writech(MFC_WrTagMask, 1 << TAGID);

spu_mfcdma64(dma_buf, eah, (unsigned)dma_list,
             sizeof(mfc_list_element_t) * LIST_NUM, TAGID, MFC_PUTL_CMD);

// DMA要求後，必ずLIST_NUM - 1回リストDMAのストールが発生するはずなので
// 先行して待つ（最後の要素はストールしないので）
for (i = 0; i < LIST_NUM - 1; i++) {
    spu_readch(MFC_RdListStallStat);
    // 特に何もせずにストールを解除
    spu_writech(MFC_WrListStallAck, TAGID);
}
spu_mfcstat(MFC_TAG_UPDATE_ALL);
```

GCC がクラッシュするプログラム

リスト 10.7 のどこか適当なところに，`dma_list[0].notify` にリテラルで 1 を代入する行を書き足して `spu-gcc43` でコンパイルしようとすると，コンパイルの途中で GCC がクラッシュします．おそらくリストの構造体にビットフィールドが使用されているためだと思われるのですが …．

10.4 アトミック更新

もう 1 つ，MFC が提供している特殊な DMA 転送を紹介します．SPE 上から主記憶のアトミック更新を行う DMA 転送です．

10.4.1 アトミックな更新と LL/SC 命令

アトミック更新自体，少々概念的にわかりにくい話なので，最初に SPE の DMA 転送に限らない一般的な話をします．

まず，アトミックな更新そのものについてですが，これはメモリ共有型のマルチプロセッサ環境などにおいて，あるプロセッサがメモリを読み出して変更して書き戻すまでの一連の操作を，他のプロセッサに邪魔されずに完結するための機能です．この機能がない (あるいはあって

も利用しない) と，複数プロセッサで共有変数を更新しようとした場合に，予期せぬおかしな動きとなってしまうことがあります．pthread で `pthread_mutex_lock` を呼ばずに共有変数を読み書きするようなコードを思い出せば，何が起こるか想像がつくことでしょう．逆に，この機能があると，これを利用して複数プロセッサ間でのソフトウェアレベルでの排他制御 (セマフォやミューテックスなど) を簡単に作ることができ，安全に共有変数を更新できるようになります．

次に，このアトミック更新の実現に用いられる方式の 1 つである LL/SC 命令について説明します．PowerPC や MIPS などの RISC 系のプロセッサには，Load Linked/Store Conditional (LL/SC) 命令と呼ばれるアトミックなメモリの更新 (と実質的に等価な効果) を実現する命令のペアが用意されています．LL 命令は特殊なロード命令で，SC 命令は特殊なストア命令です．ある主記憶上の領域に対してアトミックな更新を行いたい場合は，まず LL 命令で主記憶から汎用レジスタに値をロードし，レジスタの値を更新した上で，SC 命令で汎用レジスタの値を主記憶の同じ場所へとストアします (RISC なので値をいじる際は常にレジスタに読み込む)．これだけだと普通のロード・ストア命令と何ら違いがないのですが，LL/SC 命令では，LL 命令で変数をレジスタに読み出した後に，他のプロセッサがストア操作を行って主記憶上のその変数を変更してしまった場合，その後の SC 命令でのストアが必ず失敗するという仕掛けが備わっています．この場合，ストアしようとした値はメモリに書き込まれません．ストアが成功したか失敗したかは，ストアしようとしたレジスタの中身の変化などでソフトウェア的に認識できるようになっており，失敗した場合は LL 命令からあらためてやり直します．一方，ストアに成功したときは，ロードしてからストアするまでの間にその変数を誰にも書き換えられずに済んだということになるので，アトミックな更新が行えたのと等価となります．

10.4.2　DMA 転送によるアトミック更新

続いて，SPE の DMA 転送によるアトミック更新について紹介します．SPE は，主記憶上の共有変数へのアクセスにおいて，LL/SC 相当の機能を提供する getllar/putllc と呼ばれる DMA 転送機能を持っています．SPE が，ある主記憶上の領域をアトミックに更新する場合，まず getllar (LL に相当する主記憶からの DMA 読み出し) を MFC に要求して，主記憶から LS にデータを読み出します．MFC は getllar を処理して DMA 転送でデータを読み出すと，読み出した領域に対応したリザベーションと呼ばれる情報を MFC 内に保持します．次に，SPE は，読み出した領域の内容をアップデートするために，LS 上のデータを putllc (SC に相当する主記憶への DMA 書き込み) で主記憶に書き込むことを MFC に要求します．MFC は，putllc の要求を受けると DMA 転送で主記憶への書き込みを行おうとしますが，その際，転送先の領域に対応するリザベーションが MFC の中に残っていなかったら書き込みを行わず，書き込みに失敗したことを SPU に通知します．一方，転送先の領域に対応するリザベーションが MFC の中に残っていたら，実際に主記憶へ書き込みを行います．

リザベーションは，他の SPE や PPE が，その領域に対して書き込みを行ったことを MFC が検出すると消失します*14．したがって，元々あったはずのリザベーションが消えているというこ

*14 リザベーションが MFC に存在している状態で別の領域を getllar で読み出した場合もリザベーションは消失しますが，これは他の SPE や PPE がどうこうという問題ではないのでとりあえず考えなくてよいでしょう ….

とは，getllar で読み出した後，別の誰かが値を上書きしたことを意味します*15．putllc の成否は，DMA 転送要求後にチャネルを介して確認することができます．なお，EA 空間にマップされた LS に対しては，アトミック更新はうまく機能しないようです．

10.4.3 アトミック更新の制約

getllar や putllc には以下の制約が存在します．

- 転送サイズは 128Byte 限定で，転送対象となる EA 空間上の領域と LS 上の領域は，ともに 128Byte 境界にアラインしていなければならない
- MFC に対して要求をすると，コマンドのキューにたまらずに即座に処理されるため，フェンスやバリアなどの属性を用いて他の DMA 転送などの完了を待ってから処理を開始するといったことができない
- 複数の getllar や putllc を連続して発行することができない (次のアトミック更新の要求を発行する前に必ず完了を確認する必要がある)
- タグをつけることができないため，タグを用いて他の DMA 転送やリスト DMA とあわせて完了を検知することができない

転送サイズと EA 空間上の転送対象領域のアドレスの制約は PPE のキャッシュのラインサイズからきているようですが，LS 側の領域も 128Byte 境界にアラインさせておかないと正常に動作しない点に注意が必要です．

10.4.4 アトミック更新の要求の仕方

getllar や putllc も，通常の DMA 転送などと同様に `spu_mfcdma64` や `spu_mfcdma32` を呼ぶ (あるいはチャネルに必要事項を直接書き込む) ことで MFC に要求を発行します．実際に行われるデータ転送だけを見れば，getllar は単なる 128Byte の主記憶からの DMA 読み出しであり，putllc は単なる 128Byte の主記憶への DMA 書き込みでしかないため，リスト DMA のような変な引数 (あるいはチャネル) の使い方をする必要はありません．制約を満たすサイズやアドレスを指定しつつ，MFC へ通知するコマンドをアトミック更新用のものにするだけで発行可能です*16．表 10.7 に getllar および putllc を MFC に要求するコマンドを示します．

表 10.7 getllar および putllc を要求するコマンド

コマンド名	機能
`MFC_GETLLAR_CMD`	getllar で主記憶からデータの読み出しを行う
`MFC_PUTLLC_CMD`	putllc で主記憶へデータを書き込む

これらのコマンドは，MFC のキューに投入されると即座に処理されます．すなわち，キューの中に入って他のコマンドの後に処理されたり，後からキューに入れた別のコマンドに追い抜かれたりすることがありません．ただし，MFC に要求を行う際にチャネルにコマンドを書くという点は，他の DMA 転送要求と変わりないので，要求発行する際はキューに 1 つ以上の空きが存

*15 putllc ではなく普通のストアや DMA 書き込みが行われた場合でもリザベーションは消失します．
*16 タグは使わないので何を指定してもよさそうですが，とりあえず 0 にしておくのが無難そうです．

在する (MFC_Cmd チャネルのチャネルカウンタの値が 1 以上である) 必要があります．

また，通常の DMA 転送では，spu_mfcstat を呼ぶなどしてタグごとに転送の完了を調べていましたが，前述の通り，getllar や putllc はそもそもタグを持たないため，代わりに MFC_RdAtomicStat という専用のチャネルを読むことで完了の確認をします．MFC_RdAtomicStat チャネルは，getllar や putllc などを発行していない (もしくは完了していない) 状態で読むと SPU がストールします．getllar や putllc が完了すると，中身がセットされて読み出し可能となります．深さは 1 段なので，getllar や putllc を 1 回発行するたびに必ず 1 回読んで値を抜き取る必要があります．getllar の後に MFC_RdAtomicStat チャネルを読むと MFC_GETLLAR_STATUS というフラグがセットされている状態となりますが，getllar については，処理 (読み出し) が完了したかどうかがわかればよいので，MFC_RdAtomicStat チャネルを読んで戻るのを待つだけで十分です．一方，putllc に関しては，処理が終わったかどうかということに加えて，ストアに成功したかどうかも確認する必要があります．putllc の処理が完了して，さらに結果が失敗だった場合，MFC_RdAtomicStat チャネルを読んだ結果に MFC_PUTLLC_STATUS フラグが立ちます．うまくいった場合は何のフラグも立ちません (0 が読める)．

10.4.5 アトミック更新の例

アトミック更新を行うには，主記憶上や LS 上に 128Byte アラインされたバッファを用意する必要があります．リスト 10.8 のように__attribute__((aligned(128))) をつければよいのですが，10.1.5 で述べたように，この方法では自動変数は 128Byte アラインにならないため，関数の外でグローバル変数として確保するなどの別の方法をとる必要があります．

リスト 10.8 128Byte 境界にアラインされたアトミック更新用のバッファを確保する例

```
unsigned atomic_buf[128 / sizeof(unsigned)] __attribute__((aligned(128)));
```

実際に SPE から getllar と putllc を用いて主記憶上の共有変数をアトミックに更新する例をリスト 10.9 に示します．リスト 10.9 の例では，単純に eah と eal で表される主記憶上の領域から読み出したデータをインクリメントして書き戻すことを行っています．最初の spu_mfcdma64 関数で getllar を行い，その後，すぐに MFC_RdAtomicStat チャネルを読んで完了を待っています．次に，読み出したデータをインクリメントした後，putllc で書き戻しています．

リスト 10.9 主記憶上の共有変数を SPE からアトミックに更新する例

```
    do {
        // 主記憶上の共有変数の値をgetllarで読み出す
        spu_mfcdma64(atomic_buf, eah, eal, 128, 0, MFC_GETLLAR_CMD);
        // getllarしたら必ずMFC_RdAtomicStatを読んで完了を確認
        spu_readch(MFC_RdAtomicStat);

        atomic_buf[0]++;

        // putllcで書き戻し
        spu_mfcdma64(atomic_buf, eah, eal, 128, 0, MFC_PUTLLC_CMD);

        // MFC_RdAtomicStatから読んだ値にMFC_PUTLLC_STATUSのビットが
        // 立っていたら失敗なので再度やり直し
    } while (spu_readch(MFC_RdAtomicStat) & MFC_PUTLLC_STATUS);
```

書き戻しが成功したらアトミック更新処理は完了となり，そうでなければやり直しとなることから，ここでは putllc の結果を do ループの継続条件として判定しています．putllc に失敗した場合，MFC_RdAtomicStat チャネルから読んだ値に MFC_PUTLLC_STATUS フラグがセットされます．リスト 10.9 の例では AND を取ってこれを判定していますが，Cell/B.E. なら成功時は全ビットが 0 になることが確実なので，この部分はケチってリスト 10.10 のように書いてもよいでしょう．

リスト 10.10 アトミック更新に成功したかどうかをチャネルの値だけで判定する場合

```
    } while (spu_readch(MFC_RdAtomicStat));
```

10.4.6 putlluc と putqlluc

これまでアトミック更新の DMA 転送として getllar と putllc を紹介しましたが，アトミック更新には，これらの他に putlluc と putqlluc という 2 つの処理があります．

putlluc は，putllc と同様に主記憶へ値を書き込む際に用いるのですが，putllc と異なり，現在のリザベーションの状態に関わらずストアを行うことができます．コマンドを要求したら即座に実行される点などについては putllc とまったく同様です．こうなってくると通常の 128Byte の DMA 転送による書き込みと何が違うのかと疑問に思われるかもしれませんが，得られる結果のみを考えるならば，Cell/B.E. ではほとんど違いはないものと思われます．実際やってみると分かるのですが，Cell/B.E. の場合，主記憶にメモリコヒーレンシ必須属性が最初からついているからか，getllar で読み出されている領域に対して，別の SPE から通常の DMA 転送で書き込みを行ったり，PPE からストアを行ったりするだけでも，リザベーションを消失させることができてしまいます[*17]．なお，putllc と同じように putlluc もタグがつかないことから，putlluc の発行を要求した後は，MFC_RdAtomicStat チャネルを参照して完了を確認する必要があります．その際，MFC_RdAtomicStat チャネルから読んだ値には MFC_PUTLLUC_STATUS フラグが立ちます．

また，putqlluc は，putlluc を MFC のコマンドキューに入れられるようにしたバージョンです．putlluc に限らず，これまで紹介した 3 つのアトミック更新に関連する処理 (getllar，putllc，putlluc) は，いずれも MFC のコマンドのキューに空きがあれば即座に発行でき，キューにたまることなく優先して処理が行われるものでしたが，putqlluc は他の DMA 転送などのコマンドと同じようにキューに入って処理されます．putqlluc には最初からフェンス属性がついており，先に発行された同じタグのついたコマンドよりも必ず後に処理されるようになっています．また，処理が終了したかどうかの確認も，他のアトミック更新のように専用のチャネルを使うわけではなく，通常の DMA 転送と同じように spu_mfcstat を呼んで (MFC_RdTagStat チャネルから情報を取得して) 確認します．ますます通常の DMA 転送との違いがわかりません．なお，使いどころがよくわかりませんが，putqlluc のコマンドは，キューに複数同時に投入することも可能なようです．

---

*17 細かく考えると，putllc と同様にキューにたまらずに即座に処理される点や，コマンドの処理が完了したかどうかを MFC_RdAtomicStat チャネルを読んで確認しなければならない点などの違いありますが ….

putlluc および putqlluc を要求するためのコマンドを表 10.8 に示します．

表 10.8 putlluc および putqlluc を要求するコマンド

コマンド名	機能
`MFC_PUTLLUC_CMD`	putlluc で主記憶へデータを書き込む
`MFC_PUTQLLUC_CMD`	putqlluc で主記憶へデータを書き込む

10.4.7 アトミック更新を用いたセマフォっぽい何か

せっかくなので，これまで紹介したアトミック更新の機能を利用してセマフォっぽいものを作ってみたいと思います．

セマフォについて

まずはセマフォについての一般的な話をします．セマフォはプログラムのある部分 (クリティカルセクション) に同時に入ることのできるプロセッサの数を一定数以下に制限する仕組みです．セマフォでは，共有変数 (セマフォ変数) の値を 1 減らす操作 (P 操作) と 1 増やす操作 (V 操作) を用意し，セマフォ変数の初期値を，同時に入ることを許すプロセッサ数の上限に設定しておきます (たとえば最大 3 個のプロセッサまでの同時実行を許可する場合は，セマフォ変数を 3 に設定しておく)．また，“セマフォ変数が 0 のときは，P 操作はできないので待つ” というルールを設けておきます．クリティカルセクションを実行したいプロセッサは，まず P 操作を行ってセマフォ変数の値を 1 減らしにかかります．もし P 操作がうまくいったらクリティカルセクションに入ることができます．不幸にも P 操作を行おうとした時点でセマフォ変数が 0 だった場合，そのプロセッサはクリティカルセクションに入らず，セマフォ変数が 1 になるのを待ちます．クリティカルセクションに入ることのできたプロセッサは，今度は抜ける際に V 操作を行い，セマフォ変数の値を 1 増やします．実に単純な仕組みなのですが，ここで注意すべき点があります．セマフォ変数は共有変数なので，P 操作と V 操作は，ともにアトミックにメモリを更新できなければまともにセマフォ変数の値を管理することができません．

セマフォをアトミック更新で実現してみる

アトミック更新を用いて実装した P 操作と V 操作の例をリスト 10.11 とリスト 10.12 に示します．P 操作 (関数 `p`) では，セマフォ変数 (`eah` と `eal` で表されるアドレスの領域) を getllar で読んできて，読み出したセマフォ変数が 0 以下だったら何もせずに `return` します．そうでなく，1 以上だったら，1 減らすことに挑みます．そして putllc で書き戻して，アトミックに 1 を減らすことができたら成功として，デクリメントする前のセマフォ変数の値を返します．書き戻しに失敗したら，何もせずに `return` します．

一方，V 操作 (関数 v) は，セマフォ変数を読み出して 1 増やして書き戻しますが，アトミック更新に失敗しても即座に戻らず，うまくいくまで延々と繰り返すようになっています．なお，セマフォ変数は，SPE が読み出し始める前に PPE が適切な値に初期化しておく必要があります．

リスト 10.11 アトミック更新を用いた P 操作の実装例

```
int
p()
{
    int ret = 0;
    unsigned stat;
    // getllarでセマフォ変数を読み出す
    spu_mfcdma64(atomic_buf, eah, eal, 128, 0, MFC_GETLLAR_CMD);
    spu_readch(MFC_RdAtomicStat);
    if (atomic_buf[0] > 0) {
        // セマフォ変数が0以上だったらチャンスあり
        // セマフォ変数をアトミックにデクリメントすることに挑戦
        atomic_buf[0]--;
        spu_mfcdma64(atomic_buf, eah, eal, 128, 0, MFC_PUTLLC_CMD);
        stat = spu_readch(MFC_RdAtomicStat);
        // 失敗したら0を，成功したらデクリメント前のセマフォ変数の値を返す
        ret = (stat & MFC_PUTLLC_STATUS) ? 0 : (int)atomic_buf[0] + 1;
    }
    return ret; // < 0: エラー，0: 失敗，> 0: 成功
}
```

リスト 10.12 アトミック更新を用いた V 操作の実装例

```
void
v()
{
    do {
        // getllarでセマフォ変数を読み出す
        spu_mfcdma64(atomic_buf, eah, eal, 128, 0, MFC_GETLLAR_CMD);
        spu_readch(MFC_RdAtomicStat);
        // セマフォ変数をアトミックにインクリメントすることに挑戦
        atomic_buf[0]++;
        spu_mfcdma64(atomic_buf, eah, eal, 128, 0, MFC_PUTLLC_CMD);
        // 失敗しても戻らずに，成功するまで繰り返す
    } while (spu_readch(MFC_RdAtomicStat) & MFC_PUTLLC_STATUS);
}
```

10.4.8 アトミック更新を用いたミューテックスっぽい何か

せっかくなので，ついでにミューテックスっぽいものも作ってみましょう．

ミューテックスについて

ミューテックスは，高々1 つのプロセッサだけがクリティカルセクションに入ることのできる排他制御になります．したがって，セマフォを自由に作ることのできる環境があるならば，セマフォ変数の初期値を 1 にしておくことで簡単にミューテックスを実現できます[*18]．

ミューテックスをアトミック更新で実現してみる

前述の通り，ミューテックスはセマフォ変数の初期値を 1 にすればできあがりなのですが，もう少し効率的な実装を考えてみました．まず，リスト 10.13 に示すロック処理を見てみましょう．最初に `eah` と `eal` の指す先の共有変数 (あらかじめ PPE で 1 に初期化しておく) から getllar で値を読み出して，0 でないかどうかで分岐します．0 以外だったら，誰よりも早く 0 を putllc で書き戻すことでロックを獲得できる可能性があるので，書き戻しに挑戦します．ここでうまく書き戻すことができれば，ロック獲得となりループを抜けます．そうでなかったら，もう 1 回 getllar からやり直します．ここは特に新しいことはないでしょう．

[*18] もっときちんと知りたい場合は，きちんとした本を適宜参照してください．

リスト 10.13 アトミック更新を用いたミューテックスのロック処理の実装例

```
void
mutex_lock()
{
    do {
        // getllarで共有変数を読み出す
        spu_mfcdma64(atomic_buf, eah, eal, 128, 0, MFC_GETLLAR_CMD);
        spu_readch(MFC_RdAtomicStat);
        if (atomic_buf[0]) {
            // 読めた値が非ゼロだったら0を書き戻すことに挑戦
            // アトミックに書き戻せた者だけが排他アクセスする権利をえられる
            atomic_buf[0] = 0;
            spu_mfcdma64(atomic_buf, eah, eal, 128, 0, MFC_PUTLLC_CMD);
            if (!(spu_readch(MFC_RdAtomicStat) & MFC_PUTLLC_STATUS))
                // 成功したらループから抜ける
                break;
        } else {
            // 読めた値がゼロだったら他の誰かが排他アクセス中
            // リザベーション消失イベントが発生するまでストールして待つ
            spu_readch(SPU_RdEventStat);
            // イベントを検出したら一応アクノリッジしておく
            spu_writech(SPU_WrEventAck, MFC_LLR_LOST_EVENT);
        }
    } while (1);
}
```

一方，最初に getllar した時点で共有変数が 0 だった場合，誰かが既にロックを獲得した後ということになるので，再度 getllar からやり直します．ただし，ここではすぐにやり直しに行かずに，SPU_RdEventStat というチャネルを読みに行っています．詳しくは第 14 章で紹介しますが，SPU_RdEventStat は，あらかじめビットマスクで指定しておいたイベントが発生すると，値がセットされて読み出すことができるようになるチャネルです．どんなときにイベントを発生させるかは，SPU_WrEventMask チャネルに対してビットマスクを設定しておくことで指定します．リザベーションの消失でイベントが発生するようにするには，リスト 10.14 のように，MFC_LLR_LOST_EVENT を書き込んでおきます．main 関数の中などで初期化時に行っておくとよいでしょう．

リスト 10.14 リザベーションの消失が発生するようにイベントのマスクを設定

```
spu_writech(SPU_WrEventMask, MFC_LLR_LOST_EVENT);
```

SPU_RdEventStat チャネルは中身が空，すなわちイベントが発生していないときに読みにいくと，例のごとく SPU がストールします．よって，このような形とすることで，リザベーションが消失するまで先に進まない状態を作り出すことができます．

getllar をした時点で他のプロセッサがロックを持っていたということは，そのプロセッサがきちんと動いていれば，いずれロックが解放される (すなわち共有変数に 1 が書き戻される) はずです．1 が書かれるとリザベーションが消失することになるので，それを知らせてくれるイベントが発生するまでじっと待っていれば，無駄な getllar を繰り返さずに済むというわけです[*19]．

なお，リザベーションの消失が発生すると，SPU_RdEventStat チャネルから読み出した値に MFC_LLR_LOST_EVENT フラグが立ちます．イベントを検出した後は，次にまた同じイベントが発生した際に検出できるように，内部でイベント発生状態を保持しているフラグを一旦下げるべ

[*19] 文献 [1] や文献 [5] に，バスが混むから getllar でポーリングするのはやめろと書かれています．

く，検出したイベントと同じフラグを SPU_WrEventAck チャネルに書き込んでいます[20].

ロックに関して長々と書きましたが，次にリスト 10.15 に示すアンロック処理の方を見てみましょう．アンロックはたったのこれだけです.

リスト 10.15　アトミック更新を用いたミューテックスのアンロック処理の実装例

```
void
mutex_unlock()
{
    // putllucで無条件に1を書き戻しつつ他のSPEのリザベーションを消す
    atomic_buf[0] = 1;
    spu_mfcdma64(atomic_buf, eah, eal, 128, 0, MFC_PUTLLUC_CMD);
    spu_readch(MFC_RdAtomicStat);
}
```

ロック解放のための共有変数への書き込みは，ロックを獲得している SPE からしか呼ばれないため，絶対に書き込みが競合することがありません．したがって，ここではロック獲得時のようなアトミックな更新を行う必要がなく，いきなり putlluc で 1 を書いて終わりにしています．また，ここでは putlluc の代わりに putqlluc を利用することも可能です．アンロックする直前に主記憶へ DMA 書き込みを行う処理が入るような場合は，その処理と同じタグで putqlluc を後から発行することで，DMA 転送の完了待ちを 1 回減らせることになり，より効率的になります.

10.4.9　PPE を交えたアトミック更新

これまで紹介したアトミック更新の使用例は，SPE 同士の間での競合を防ぐためのものでした．しかしながら，場合によっては，PPE からも共有変数を更新したいということもあるでしょう．そのような場合，PPE の LL/SC 命令である lwarx と stwcx という 2 つの命令を利用して共有変数にアクセスします．これにより，getllar/putllc を用いて共有変数にアクセスを行っている SPE に混ざって，PPE からも共有変数のアトミックな更新が行えるようになります．PPE 側でこれら命令を利用するにはインラインアセンブラを利用するのが一般的なようですが[21]，ppu-gcc の場合，型固有組み込み関数を用いて呼び出すことができるため，通常の C 言語の関数と同じ感覚で利用することができます．以下ではその方法を簡単に紹介します.

PPE で型固有組み込み関数などを利用する場合は，ppu_intrinsics.h を include します.

PowerPC の LL 命令に相当する lwarx を呼び出す型固有組み込み関数は__lwarx です.

```
d = __lwarx(p);
```

上記のフォーマットの場合，引数 p にアトミック更新したい領域を指すポインタを渡すと，リザベーションが作られ，d にポインタの指す先の領域の先頭 1 ワードの中身が格納されます.

また，PowerPC の SC 命令に相当する stwcx は__stwcx で呼び出します.

```
d = __stwcx(p, a);
```

上記のフォーマットの場合，lwarx で読み出したアドレス p に対して a の値をストアすることに

[20] 理由は不明ですが，リザベーションの消失に関しては，これをやらなくても再度検出できるようです.

[21] ググって出てくるのはインラインアセンブラを使った例が多いので ….

なります．ストアが成功すると d に 1 が格納され，そうでない場合は 0 が格納されます．

というわけで，これらをリスト 10.16 のように記述することで，PPE によるアトミック更新が可能となります．リスト 10.16 の atomic_buf を，SPE が DMA 転送でアトミック更新する領域にすれば，SPE 間で行っている getllar/putllc によるアトミックな更新に PPE が混ざることが可能となります．

リスト 10.16　PPE による共有変数のアトミック更新

```
unsigned d;
do {
    // atomic_bufから読み出し
    d = __lwarx(atomic_buf); // dには*atomic_bufが入る
    d++;
    // atomic_bufへの書き戻しに失敗したら読み出しからやり直す
} while (!__stwcx(atomic_buf, d));
```

なお，PPE が lwarx でリザベーションを作ることのできるアドレスには制限はないようなのですが，SPE と排他アクセスをするのであれば，128Byte アラインさせた領域の先頭を利用せざるを得ないでしょう．

10.5　LS 間の DMA 転送

これまでも何度か述べてきたように，各 SPE の LS は EA 空間にマップされるため，PPE からロード・ストア命令を用いて直接アクセスすることができます．また，SPE からは MFC による DMA 転送機能を用いて EA 空間に対してアクセスできるため，PPE から見えているのと同じ EA に DMA 転送でアクセスすることで，他の SPE の LS にアクセスすることが可能です．これにより，SPE は，自分以外の他の SPE の LS を DMA 転送で読み書きすることができます[*22]．SPE を同時に複数利用する並列処理では，主記憶を介して SPE 同士が通信をしてもよいのですが，LS 間で直接 DMA 転送をした方が効率的な場合もあります．ここでは SPE が自身の LS と他の SPE の LS との間で DMA 転送を行うための方法を紹介します．

10.5.1　LS の EA を取得する

各 SPE は，自分や他の SPE の LS が EA 空間のどこにマップされているかを知る術を持たないため，SPE が他の SPE の LS にアクセスを行うには，あらかじめ PPE が SPE に対して他の SPE の LS の EA を教えておく必要があります．PPE のユーザプログラムからは，libspe2 の API を呼び出すことで各 SPE の LS の EA を取得することができるので，これを SPE に通知しておくことで，SPE が自身の LS と他の SPE の LS との間で DMA 転送できるようになります．LS の EA を取得するための libspe2 の API を以下に示します．

spe_ls_area_get：LS の EA を取得する

```
void *spe_ls_area_get(spe_context_ptr_t spe);
```

[*22] 厳密には LS は一部の属性が主記憶と異なるため，細かいところで主記憶に対するアクセスと違う部分があるのですが，ひとまず気にせずに話を進めます．

spe_ls_area_get は，spe で指定したコンテキストの LS の EA を返します．エラーの場合は NULL が返り，errno にエラーコードがセットされます．

リスト 10.17 に spe_ls_area_get で LS の EA を取得する際の例を示します．

リスト 10.17 spe_ls_area_get の使い方の例

```
void *ls;
spe_context_ptr_t context;
...
context = spe_context_create(SPE_MAP_PS, NULL);
...
ls = spe_ls_area_get(context);
```

リスト 10.17 では，ls に LS の開始アドレスを格納しています．このようにして取得したポインタの指す先を PPE 上でアクセスすることで，PPE のプログラム内から LS を読み書きすることが可能となり，また，ここで得た EA を SPE に通知することで，SPE がその LS に対して DMA 転送でアクセスできるようになります．

ただし，ここで気をつけなければならない点が 1 つあります．SPE や PPE が他の SPE の LS の EA を知ったとしても，それだけではまともな DMA 転送を行うことはできません．API を介して知ることができるのは，あくまでも LS の先頭に対応するアドレスでしかありません．一方，実際に SPE プログラムが参照したいのは変数や配列として LS 上に確保されている領域です．そのため，LS 間で DMA 転送を行うには，LS の先頭 EA だけでなく，転送相手となる SPE において，データのやりとりに使う変数や配列が LS の先頭からどれくらいのオフセットの位置に確保されているのかという情報も必要になります．

SPE が別の SPE の LS に対して DMA 転送を行う場合，両者が同一の実行バイナリを動かしていて，さらにグローバル変数などのアドレスが最初から決まっている領域を DMA 転送の対象とすれば，どの SPE で動かしても LS 上のその領域のアドレスは同一となるため，他の SPE の DMA 転送対象領域の EA を確実に求めることができます (図 10.15)．

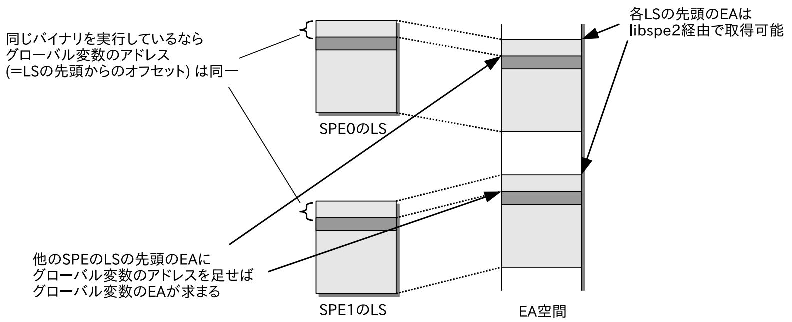

図 10.15 他の SPE の LS 上の変数の EA を求める

これができない場合は，DMA 転送を行う前に，DMA 転送で読み書きされる領域の LS 上のアドレスを何らかの方法で相手に通知しておく必要があるでしょう．

10.6 PPE からの DMA 転送の利用

libspe2 には，PPE から SPE の MFC に対して DMA 転送要求を発行するための API が用意されています．しかしながら，10.5 で述べたように，各 SPE の LS は EA 空間にマップされているため，DMA を使わなくても，ロードやストアなどの PPE の命令で通常のメモリ領域と同様に LS を読み書きすることができてしまいます．そのため，単に LS のデータを読み書きするのであれば，この API を使う理由はなさそうです (PPE のキャッシュを汚染せずに LS にデータを流し込めるという利点はあるかもしれない) が，もしかしたら Cell/B.E. 外部の I/O デバイスなどと組み合わせると面白いことができたりするのかもしれないので，ここで紹介しておきます．

10.6.1 libspe2 に用意されている DMA 転送要求発行機能

libspe2 から (と言うより SPE の外部から) MFC に対して発行できる DMA 転送要求は，これまで紹介した DMA 転送の中では，通常の読み出し (GET) と書き込み (PUT) およびそれらにフェンス・バリア属性がついたものに限られます．リスト DMA やアトミック更新を SPE の外から要求することはできません (おそらくハードウェア的に非対応)．

一方で，外部からしか要求できない DMA 転送に，スタート属性のついた DMA 転送というのがあります．これは，DMA 転送が完了すると同時に SPE の動作を開始させるという特殊な機能を提供する属性なのですが，何の役に立つのかイマイチよくわかりません*[23]．

また，MFC は，チャネルを介した SPU からの DMA 転送要求を保持するキューとは別に，外からの DMA 転送要求を受け付けるための専用のキュー持っており，同じ SPE 上の SPU から発行された DMA 転送要求と，PPE などの外部から発行された DMA 転送要求を区別して扱うようになっています．Cell/B.E. の場合，外部から発行された DMA 転送要求は最大 8 個まで同時に受け付けることができます．

spe_mfcio_put：LS から EA 空間への DMA 転送を要求

```
int spe_mfcio_put(spe_context_ptr_t spe, unsigned lsa, void *ea,
                  unsigned size, unsigned tag, unsigned tid, unsigned rid);
```

spe_mfcio_put は，spe で指定した SPE のコンテキストの MFC に対して，その SPE の LS から EA 空間への DMA 転送を要求します．名前が PPE から見たデータの転送方向と逆向き (“put” なのに読み出しとなる) に見えますが，これは API 名が MFC のコマンド名をベースとしているためです．

lsa は LS 上の転送データの開始アドレスです．PPE から DMA 転送を要求する場合でも，ここで指定するのは EA 空間にマップされた LS のアドレス (EA) ではなく，SPU や MFC から見た LS のアドレスになります．一方，ea は EA 空間上の受信領域を指すポインタになります．

---

*[23] libspe2 のソースを見ても，API どころかコマンドの定義すらヘッダにないため，この機能を試すには libspe2 の API に頼らずに，第 12 章を参考に SPE のプロブレムステート領域を libspe2 を介さずに直接アクセスして自力で DMA 転送要求を MFC に書き込む必要があります．スタート属性は，MFC_CMD_PUT や MFC_CMD_GET に 0x8 を OR することで付加することができます．

本関数を呼ぶことで，これらの領域の間での，size で指定した分の DMA 転送が要求されます．なお，これら引数は，libspe2 内でほぼそのままの状態でハードウェアのレジスタに書き込まれることになるため，10.1.2 で示した SPU からの DMA 転送要求における制約と同じ制約が付随する点に注意が必要です．

また，tag に関しては，ハードウェア的には PPE からの DMA 転送要求でも 0 から 31 までを使用可能なのですが，16 から 31 まではカーネル用に予約されているらしく，指定した場合，libspe2 がエラーを返します．ここでは 15 以下を使うようにする必要があります．

tid と rid については，おそらくほとんど無意味なパラメータかと思われるので，ここでは説明を飛ばします．何を与えても特に問題ないはずですが，256 以上の数を指定すると libspe2 に弾かれるので注意してください．これらがどのようなものなのかについては別途コラムで簡単に紹介します．

本関数によって DMA 転送の要求に成功すると 0 が返り，エラーが発生した場合は −1 が返って errno に原因を示す番号が入ります．

spe_mfcio_putb・spe_mfcio_putf：LS から EA 空間への DMA 転送を属性をつけて要求

```
int spe_mfcio_putb(spe_context_ptr_t spe, unsigned lsa, void *ea,
                   unsigned size, unsigned tag, unsigned tid, unsigned rid);
int spe_mfcio_putf(spe_context_ptr_t spe, unsigned lsa, void *ea,
                   unsigned size, unsigned tag, unsigned tid, unsigned rid);
```

spe_mfcio_putb と spe_mfcio_putf は，それぞれバリア属性とフェンス属性のついた LS から EA 空間への DMA 転送を要求します．引数や戻り値に関しては spe_mfcio_put と同様です．

spe_mfcio_get：EA 空間から LS への DMA 転送を要求

```
int spe_mfcio_get(spe_context_ptr_t spe, unsigned lsa, void *ea,
                  unsigned size, unsigned tag, unsigned tid, unsigned rid);
```

spe_mfcio_get は，EA 空間から LS への DMA 転送を要求します．spe_mfcio_put と同様に，MFC のコマンド名がベースとなっているため，PPE から見た転送方向と逆向きの名前がついています．転送方向が逆向きな点以外は，引数の意味なども含めて spe_mfcio_put とまったく同様です．

spe_mfcio_getb・spe_mfcio_getf：EA 空間から LS への DMA 転送を属性をつけて要求

```
int spe_mfcio_getb(spe_context_ptr_t spe, unsigned lsa, void *ea,
                   unsigned size, unsigned tag, unsigned tid, unsigned rid);
int spe_mfcio_getf(spe_context_ptr_t spe, unsigned lsa, void *ea,
                   unsigned size, unsigned tag, unsigned tid, unsigned rid);
```

spe_mfcio_getb と spe_mfcio_getf は，それぞれバリア属性とフェンス属性のついた EA 空間から LS への DMA 転送を要求します．引数や戻り値に関しては spe_mfcio_get と同様です．

spe_mfcio_tag_status_read：DMA 転送の処理完了を確認

```
int spe_mfcio_tag_status_read(spe_context_ptr_t spe, unsigned mask,
                              unsigned behavior, unsigned *tag_status);
```

spe_mfcio_tag_status_read は，spe で指定したコンテキストの SPE の MFC に対して DMA 転送要求の処理の完了を問い合わせて返します．mask はタグマスクに相当する引数で，転送完了状況を問い合わせたい DMA 転送要求のタグに相当するビットに 1 を立てた値を渡します．ただし，mask に 0 を指定した場合は特殊な扱いとなり，過去にその MFC に対して行ったすべての DMA 転送の完了状況を問い合わせることになります[*24]．behavior には，どのようにこの関数から戻るかを指定します．表 10.9 に behavior に指定可能な値と，それに対応する関数から戻る条件を示します．

表 10.9 behavior の値と関数からの戻る条件

値	関数から戻る条件
SPE_TAG_ALL	mask で指定したタグに対応するすべてのタググループの全 DMA 転送が完了したら戻る
SPE_TAG_ANY	mask で指定したタグのどれか 1 つに対応するタググループの全 DMA 転送が完了したら戻る
SPE_TAG_IMMEDIATE	mask で指定した DMA 転送の現時点での完了状況を返す

読み出した完了状況は，ポインタ tag_status の指す先に格納されます．本関数は成功すると 0 を返し，エラーがあった場合は −1 が返って errno にエラーの番号が設定されます．

DMA 転送のタグと一緒に指定する tid と rid

文献 [2] などを見ると，MFC_Cmd チャネルには，MFC コマンドの他に tid と rid という 2 つの謎の ID を合わせて書き込める仕様となっています．ドキュメントを見ても “tid は転送クラス ID” で “rid は置換クラス ID” としか書かれていない上に，空いているビットに無理矢理ねじ込んだ感じで配置されているため非常に気持ちが悪いのですが，それぞれがどのようなものなのかについては，一応文献 [5] などに記載があります．

tid の方は “TclassID” の略で，バスのクォータ (帯域制限) の選択に使います．この ID をうまく指定することで，転送ごとにバスの帯域の割り当てを調整することができるようになります．ただ，この機構はデフォルトで無効化されているようで，特権ソフトウェアが有効化しないと使えないみたいです．文献 [7] を見ると，この機構を有効にするためのレジスタはハイパーバイザからのみ触ることができるようなところにあるらしく，PS3 Linux においてこのレジスタが設定されているのかどうかは不明です．また，この機能は CBEA では実装依存とされている機能なため，Cell/B.E. に搭載されているのかどうかもよくわかりません．

一方，rid の方は “RclassID” の略で，PPE の L2 キャッシュや TLB のエントリのリプレースに影響します．どのエントリを追い出すかなどを指定するのですが，こちらに関連するレジスタについてもハイパーバイザのみがアクセスできるような特権領域にあり，また CBEA では実装依存とされていることから，やはりハードウェア的に利用できるのかどうかすらわかりません．

[*24] ハードウェア的にこのような機能を提供しているわけではなく，単に libspe2 の中で過去に発行して完了を確認していない DMA 転送要求のタグを記憶していて，それを利用して実現しています．

10.6.2 libspe2 から DMA 転送を行う例

リスト 10.18 に，PPE から libspe2 で MFC に DMA 転送要求を発行する際の PPE の処理を示します．最初に DMA 転送用のバッファ (`dma_buf`) を初期化し，次に SPE から送信メールボックスに値が書き込まれるのを待ちます．ここでは SPE によって LS の DMA 転送データのアドレスが書き込まれるのを待っています．アドレスが書き込まれたら，`spe_mfcio_put` で LS から PPE 側の `dma_buf` へのデータの DMA 転送を要求し，その後，`spe_mfcio_tag_status_read` で完了を待ちます．このとき，PPE は MFC のキューの空きを確認していませんが，実際には `spe_mfcio_put` を呼んだ際に，libspe2 内部でキューに空きができるまでループして待つことをしています[*25]．DMA 転送が完了したら，確認のために値を出力し，受信に用いた PPE 側のバッファの値を書き換えて，今度はそこから LS へデータを転送するよう DMA 転送要求を発行して完了を待っています．

リスト 10.18 PPE から DMA 転送を要求する際の PPE 側の処理の例

```
#define TAGID 3
    ...
    for (i = 0; i < sizeof(dma_buf) / sizeof(dma_buf[0]); i++)
        dma_buf[i] = 0;

    // 送信メールボックスでSPEのバッファのアドレスを受け取る
    while (spe_out_mbox_read(spethreads.context, &mbox_data, 1) < 1)
        ;

    // LS上のバッファからdma_bufへ16Byte書き込ませる要求発行
    spe_mfcio_put(spethreads.context, mbox_data, (void *)dma_buf, 16,
                  TAGID, 0, 0);
    // 転送が完了するまで待つ
    spe_mfcio_tag_status_read(spethreads.context, 1 << TAGID,
                              SPE_TAG_ALL, &tagstat);

    printf("PPE: dma_buf[0] = 0x%08x\n", dma_buf[0]);
    dma_buf[0] = 0xcbea0123;

    // dma_bufからLS上のバッファへ16Byte読み出させる要求発行
    spe_mfcio_get(spethreads.context, mbox_data, (void *)dma_buf, 16,
                  TAGID, 0, 0);
    // 転送が完了するまで待つ
    spe_mfcio_tag_status_read(spethreads.context, 1 << TAGID,
                              SPE_TAG_ALL, &tagstat);
```

一方，これに対する SPE 側の処理の例をリスト 10.19 に示します．最初に待ち合わせを兼ねて，LS の DMA 転送領域のアドレスを送信メールボックス経由で PPE に伝えています．10.5 でも述べましたが，PPE 側からは SPE のプログラムが確保したバッファのアドレスを知ることはできないので，SPE に教えてもらう必要があります．アドレスを送信メールボックスで通知した後は，PPE から DMA 転送要求が発行されて DMA 転送領域の値が更新されるのを待ちます．

大した処理ではありませんが，このようにすることで外から DMA 転送を仕掛けることができます．ちなみに，libspe2 が DMA 転送の要求を発行したり，処理が完了したことを確認したりする際は，プロブレムステート領域と呼ばれる領域に含まれる MFC のレジスタに対してアクセスを行っています．これらはすべて EA 空間にマップされているため，各 SPE から DMA 転送を利用してこの領域にアクセスすることで，SPE から別の SPE の MFC にアクセスして DMA

[*25] ライブラリの実装としてそれはどうなんだという気がしなくもないですが ….

リスト 10.19 PPE から DMA 転送を要求する際の SPE 側の処理の例

```
volatile unsigned dma_buf[4] __attribute__((aligned(16)));

dma_buf[0] = 0x89abce11;

// LS上のバッファのアドレスを送信メールボックスで通知
d = (unsigned)dma_buf;
spu_writech(SPU_WrOutMbox, d);

// バッファの値が書き換わるまで何もせずに待機
while (dma_buf[0] == 0x89abce11)
    ;
printf("SPE%d: dma_buf[0] = 0x%08x\n", speid, dma_buf[0]);
```

転送を要求するといったことも可能となります．プロブレムステート領域に関しては，DMA 転送以外の話題も絡んでくるため，第 12 章にて扱います．

10.7 DMA 転送の最適化

10.7.1 DMA 転送の完了を待つ時間を減らす

SPE には，動画のような大容量のデータをストリーミング処理するというような使い方がありますが，そのような場合，処理対象のデータは LS に置けるような大きさではないことから，主記憶との間で絶えず DMA 転送でデータの受け渡しを行いながら処理を進める必要が生じます．その際，DMA 転送でのデータ読み出し (あるいは書き込み) 要求を行ってから，実際にデータの転送が完了して LS 上の領域のデータを安全に読み出せる (あるいは書き潰せる) ことが保証されるまで SPU を何もせずに待機させるような実装は極力避けるべきです．せっかく MFC は SPU と独立して並列動作可能なのですから，SPU と MFC が同時に処理を進められるように実装しましょう．

図 10.16 に，効率の悪い DMA 転送のシーケンスの例を示します．この例では，DMA 転送を要求した直後に，そのデータの受信が完了するのを `spu_mfcstat` で待っているため，MFC が DMA 転送を処理し終えるまで SPU は何もできず，無駄な時間を過ごすことになります．

このような場合，受信データが不要となった (受信データ保持用のバッファを上書きしてもよくなった) 後にもまだ演算処理が残っているのであれば，残りの処理を行う前に，次のデータの DMA 転送の要求を発行しておくことで，残りの処理を行っている間に MFC が DMA 転送を行い，次のデータの処理を開始する時点では必要なデータが LS 上に揃っているという状況を実現することができます*26.

あるいは処理の内容的にそのようなチューニングが難しいようであれば，受信バッファを倍の容量確保 (あるいはデータの処理単位を元々の半分に分割) するダブルバッファリング化を行い，片方のバッファ上のデータを処理している間に，もう片方のバッファへの次に処理するデータの読み出しを MFC に要求するようにデータ転送部分を変更することで，DMA 転送の完了を待つ時間を排除することができます．もちろん，元々データ転送時間に対して処理時間が極端に短い

*26 もちろん，このような場合でも，後々受信したデータを使う直前に，念のため `spu_mfcstat` を呼んで DMA 転送が終わっていることを確認する必要があります．

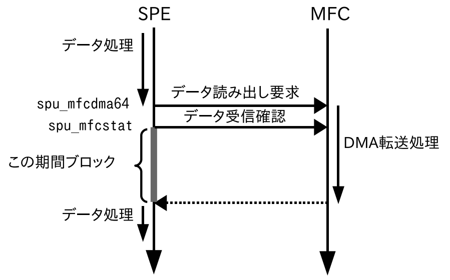

図 10.16　効率の悪いデータの処理の例

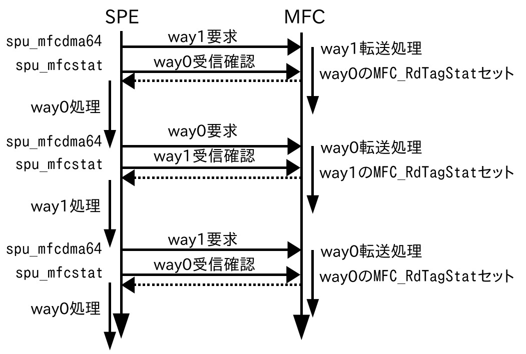

図 10.17　ダブルバッファリング化したデータ処理の例

場合は，これを行っても演算処理後に待ち時間が発生してしまいますが，それでも何もしないよりは効果があるはずです．

図 10.17 に，ダブルバッファリング化した DMA 転送のシーケンスの例を示します．図の way0 と way1 は，2 つに分けたデータ受信バッファの各々に対応します．way0 を DMA 転送で読み出す場合と way1 を DMA 転送で読み出す場合とでは，異なるタグを指定するようにします*27．うまくはまれば，DMA 転送の待ち時間をほぼ 0 にすることができます．

10.7.2　完了を確認する必要のない DMA 転送

DMA 転送の要求を spu_mfcdma64 などで発行した後，通常はどこかのタイミングで転送が完了したことを spu_mfcstat などで確認したりしますが，DMA 転送と処理の関係次第では，自身の発行した DMA 転送要求の完了を待つ必要がないケースも考えられます．たとえば非常に極端な例ですが，リスト 10.20 に示すような，各 SPE が LS 間で DMA 転送を行ってフラグを受け渡しすることでバリア同期を行うコードでは，DMA 転送の完了を確認する処理を一切必要としません*28．

リスト 10.20　DMA 転送の完了の確認を必要としない DMA 転送を含むバリア同期

```
#define SPE_NUM 6
static unsigned long long ls_addr[SPE_NUM] __attribute__((aligned(16)));
static volatile unsigned b[6][128 / sizeof(unsigned)] = {{0}};
    ...
void
barrier()
{
    if (__builtin_expect((speid == 0), 0)) {
        // SPE0は，それ以外のSPEからDMAでbの要素が更新されるのを待つ
        for (i = 1; i < SPE_NUM; i++) {
            while (!b[i][0])
                ;
            b[i][0] = 0;
        }
        // 全SPEからの更新を確認したら，各SPEにDMAで解除通知
        for (i = 1; i < SPE_NUM; i++) {
            ealsa = ls_addr[i] + (unsigned)b;
```

*27　一緒にすると，処理前に発行した DMA 転送が終わっていることを念のため確認するはずが，次の処理が終わった後に確認すればよい DMA 転送の完了までその場で待ってしまうことになるため，ダブルバッファリング化した意味がなくなってしまいます．どうしても単一のタグで済ませたい場合は，way0 の受信の完了を待ってから way1 の要求を発行するといった風に，順番を入れ替えます．

*28　アドレスの計算などに無駄が多く，最適化の余地が多々ありますが，とりあえずその辺は無視してください ⋯．

```
            eah = ealsa >> 32;
            eal = (unsigned)ealsa;
            spu_mfcdma64(&b[0][0], eah, eal, 128, TAGID, MFC_PUT_CMD);
        }
        // spu_mfcstat(MFC_TAG_UPDATE_ALL); // 不要
    } else {
        // SPE0以外のSPEはSPE0のbにDMAで値を書き，バリア同期への到達を通知
        ealsa = ls_addr[0] + (unsigned)&b[speid][0];
        eah = ealsa >> 32;
        eal = (unsigned)ealsa;
        spu_mfcdma64(&b[speid][0], eah, eal, 128, TAGID, MFC_PUT_CMD);
        // spu_mfcstat(MFC_TAG_UPDATE_ALL); // 不要
        // SPE0から解除通知が書かれるのを待つ
        while (!b[0][0])
            ;
        b[0][0] = 0;
    }
}
```

リスト 10.20 の `ls_addr` は，各 SPE の LS の EA 空間上のアドレスを格納した配列です．あらかじめ，プログラムの最初で `b[speid][0]` にだけ 1 を代入してあるものとします．すべての SPE が，この `barrier` 関数を呼び出すことで，すべての SPE の間で同期を取ることができます．`speid` が 0 以外の各 SPE は，同期を取る必要が生じたら自身の `b[speid][0]` の値 (1) を `speid` が 0 の SPE の `b[speid][0]` に DMA 転送で書き込みます．`speid` が 0 の SPE は，`b[1][0]` から `b[5][0]` までに 1 が揃ったら，それらをゼロクリアして `b[0][0]` の値 (1) を各 SPE の `b[0][0]` に DMA 転送で書き込みます．`speid` が 0 の SPE は，5 つの SPE に対しての DMA 転送が完了したら同期完了とし，それ以外の SPE は，`b[0][0]` に 1 が書かれたことを検出したら，そこをゼロクリアして，同期完了として処理を抜けます．

なぜこの処理で DMA 転送の完了を待つ必要がないのか，是非考えてみてください[*29]．些細な差かもしれませんが，思わぬところで性能差が出たりするので，このように省けるものは省いた方がよいでしょう．

複数の同じ DMA 転送

バスの混雑などの事情により要求された DMA 転送が即座に処理できないという判断を MFC が下した場合，どうも MFC は他に処理できる DMA 転送要求があればそれを処理するものの，他に処理するものがない場合は，ただただ一定時間が経過するのを待つということをしているように思われます．この一定時間が結構長いようで，バスが大混雑している状況でデータを送ると，運次第で相当待たされる場合があります．サイズの大きな転送であればスループット的にはあまり影響がないため気にする必要はないのかもしれませんが，レイテンシには大きく影響するため，データのサイズが小さい場合に非常に不利に働く可能性があります．この問題を回避する方法として，同じデータを何個も連続して DMA 転送要求する方法が考えられます．1 個だけだと後回しにされた場合に再度順番が回ってくるのを長々と待つしかありませんが，複数個あれば，それだけ一発で出て行く可能性が高まるというわけです．この辺は完全に経験則による話で，実際に混雑時の待ち方などが上記の通りなのかどうかも定かではないのですが，DMA 転送に要する時間が安定せず，処理を実行するたびに SPE 間の同期などに要する時間がぶれるような場合は，試してみる価値があるかもしれません．

[*29] べ，別に説明が面倒くさいから省いたとかそういうわけじゃないですよ ….

第 11 章
プログラムの並列化とデータ転送のチューニング

交代だ!! サンシャイン
あーっと サンシャインとザ・ニンジャ
空中で入れかわったーっ!!
— ゆでたまご『キン肉マン』より —

本章では，第 6 章で散々最適化を行ってきた Hack the Cell 2009 の課題プログラムを題材に，SPE プログラムの並列化と，そこでのデータ転送のチューニングの実例を紹介します[*1]．

11.1　SPE プログラムの並列化

一般に，プログラムを並列化する場合，プログラムが対象とする処理の内容や扱うデータの規模，要求されるスケーラビリティなどの，さまざまな要因によって並列化の手法が異なってくるため，これまで紹介した分岐やループの最適化などのように “こうすればできる” ということを系統立てて示すのは非常に困難です．そのため，本章は，Hack the Cell 2009 の課題プログラムに対して，筆者が並列化手法を考案してから適用する際の流れを天下り的に紹介する形となっています．非常に押し付けがましい感じで恐縮なのですが，“こういう順番で変更を施していけばできる” というようなものではなく，あくまでも並列化を行うまでの流れの一例として捉えてください．

11.1.1　並列化の方針決定

最初に並列化の方針について考えます．これまで散々最適化を行ってきたことからもわかるように，このプログラムの実行時間の大部分は `genrand_mine` という関数が占めています．したがって，この関数での処理をうまく複数 SPE に割り振って並列処理させることで，効果的に全体の実行時間を短縮できると考えられます．どのように処理を分割するかについては色々と考えられますが，ここでは処理の切れ目がわかり易いことから，`mt` の更新処理と Tempering 処理をそれぞれを異なる SPE に担当させることで処理を 2 つの SPE で手分けして行う，すなわち処理

[*1] Hack the Cell 2009 のコンテスト自体はチャネルを利用することが禁止されていたため，DMA 転送などを利用した並列化による最適化は行えないルールでした．

を 2 並列化することにします.

11.1.2　共有データの配置の検討

以下では説明のために mt の更新処理を行う方の SPE を “SPE0”，Tempering 処理を行う方の SPE を “SPE1” と称します．プログラムの処理を mt の更新処理と Tempering 処理に分けて行う場合，SPE0 が mt を更新し，その結果を SPE1 が読み出して Tempering 処理を行って乱数を足し込むという流れになります．したがって，SPE0 と SPE1 の間で mt を共有することになります．その際，mt をどこに置くかというのが問題となります.

SPE 間でデータを共有する場合，データは主記憶上に置くか，あるいはどちらかの LS に置くことになります．サイズの大きなデータを処理する場合や，複数の SPE で同時に読み書きを行うようなデータの処理の仕方をする場合，共有データを主記憶上に置いて，各 SPE が必要な分を適宜そこから DMA 転送で読み書きして利用する形をとることがよくありますが，今回扱う mt 自体は元々LS に収まる程度の非常に小さなサイズであり，また共有データにアクセスする SPE の数も全部で 2 個しかないので，ここでは LS 上に置いて共有することにします.

次に，どちらの SPE の LS にデータを置くかについて考えてみます．mt を SPE0 と SPE1 のそれぞれがどれだけ参照するかという点を考えてみると，SPE0 は更新処理において mt の読み書きを行うのに対し，SPE1 は Tempering 処理のために読み出しのみを行います．また，mt の 624 要素をすべて更新するには mt の各要素を何度もアクセスしなければならないのに対し，mt の 624 要素に対して Tempering 処理を行う場合，各要素を 1 回読み出すことができればそれで済みます．したがって，アクセスの頻度やアクセス内容からすると，mt は SPE0 側に置いておき，SPE1 の LS にはそのコピーをキャッシュするのが効率的だと考えられます.

SPE0 の LS から SPE1 の LS へのデータのコピーは，SPE0 か SPE1 のどちらかの MFC を利用して行うことになります[*2]．SPE0 が mt の更新を終えた直後に mt を SPE1 に DMA 転送で書き込むのと，SPE1 が SPE0 の mt の更新が完了したことを確認して mt を自身の LS に読み出すのとでは，前者の方が圧倒的に通信に要する時間が少なくて済むため，mt のコピーは SPE0 側から行うことにします．その際，SPE1 は，mt を見ているだけでは，SPE0 によってデータが上書きされたことを検知するのが難しいため，あわせて SPE0 が SPE1 に mt を更新したことを通知するための共有変数も SPE1 側に用意しておきます．SPE0 は，更新後の mt を DMA 転送で SPE1 に送った後，SPE1 上の共有変数に DMA 転送でフラグをセットします．SPE1 は，共有変数のフラグがセットされるのをポーリングして待ち，フラグがセットされたことを検出したら，mt が更新されたと判断して，Tempering 処理を開始します[*3].

また，このように，SPE0 から SPE1 にコピーを行う場合は，SPE0 側で SPE1 が前のデータの処理を終えたかどうかを把握する仕組みが必要となります (これをやらないと SPE1 が処理途中のデータを SPE0 が次々と上書きしてしまうことになる)．LS を用いてこの仕組みを実現する

[*2] PPE が EA 空間上にマップされた LS の中をロード・ストア命令で読み書きしてコピーするという方法も考えられないこともありませんが，今回は PPE は並列処理に加えない方針なので，この方法は用いないことにします.

[*3] 末尾の要素の値が書き換わるのを見張る方法も考えられますが，運悪く更新後の mt でも末尾の 1 要素の値が変わらないという場合も考えられるので，ここではその方法は用いません．そもそも，DMA 転送ではアドレス順にデータが書き込まれる保証があったかどうか ….

場合も，SPE0 が SPE1 の変数を DMA 転送で読み出して取得する方法と，SPE1 が SPE0 の変数へ DMA 転送で書き込んで通知する方法の 2 種類が考えられますが，前者だと SPE0 が毎回 mt を更新するたびに DMA 転送で SPE1 の変数をポーリングすることになり，通信量が増えて効率が悪くなることが考えられるので，ここでも後者の方法を採用します．

一度 SPE0 が SPE1 へ mt をコピーして通知を行った後は，SPE0 は，SPE1 から “処理を完了した” という通知を受けるまで，SPE1 の処理中である (かもしれない) mt を上書きしないように，次のデータのコピーを行いません．この場合，SPE1 が SPE0 に “処理を完了した” という通知を行うまでは，SPE0 から “(次の) mt のコピーが完了した” という通知も送られてこないことになるため，SPE1 上のコピー完了通知は，SPE1 が処理完了通知を送る前であれば，SPE1 側で安全にクリアすることができます．また，SPE0 上の，SPE1 による処理完了通知のための変数に関しても同様です．

ここまでで述べた共有データの配置についてまとめたものを図 11.1 に示します．

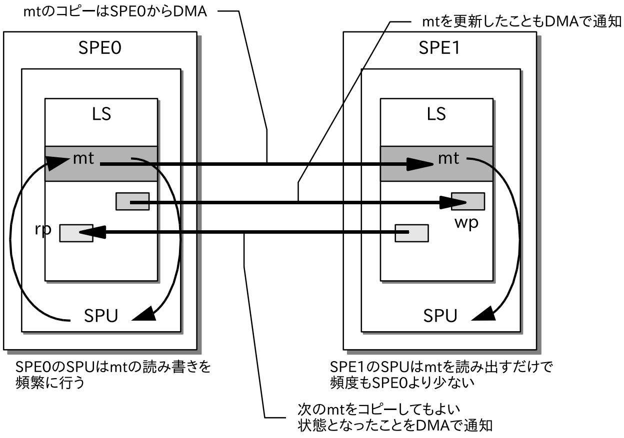

図 11.1　SPE0 と SPE1 の共有データの配置

なお，このようなデータ配置とデータのコピー方法を採用することで，SPE0 が更新処理を終えた mt を SPE1 の LS にコピーした直後から，SPE0 は次の mt の更新処理を始めることができるようになるため (Tempering 処理では mt に値を書き戻す必要がないので)，処理の並列性を高めることにもつながります．

11.1.3　並列化のための SPE プログラムへの変更

プログラムを並列化する場合，各プロセッサ向けにまったく異なるプログラムを作成して，それを別々の実行ファイルとして用意し，各プロセッサ上で異なる実行ファイルを走らせるという方法をイメージする方もいるかもしれませんが，ネットワーク上で行うサーバ・クライアント方式の分散処理ならいざ知らず，一般的な同種のプロセッサを複数利用して行う並列処理ではそういった方法はあまり用いません．このような環境では，単一のプログラムから 1 種類の実行ファイルを作成して，それを並列処理に参加するすべてのプロセッサ上で実行するのが一般的です．それだとすべてのプロセッサでまったく同じ処理を実行することになってしまうのではないかと懸念されるかもしれませんが，その心配はありません．通常，それぞれのプロセッサには異なる

ID が与えられるようになっていることから，プログラム内でその ID を取得して，その ID を元に個々のプロセッサが自発的に判断して，互いに違う処理やデータ範囲を担当するようにプログラミングをしておきます．このようなモデルでプログラミングを行うことで，開発を行いやすくなります．図 11.2 に，このような ID を用いた処理の切り分けのイメージを示します．

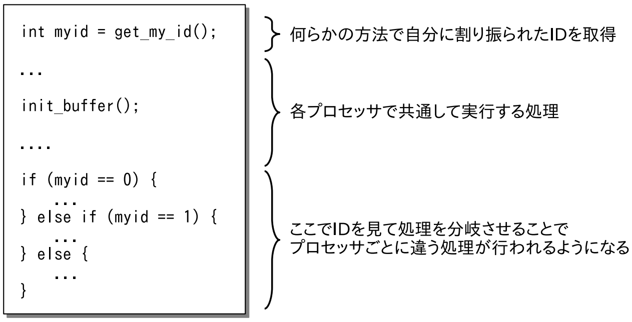

図 11.2　単一のプログラムで並列処理を行う記述のイメージ

今回の例でもこの方式を用い，2 種類のプログラムを用意したりすることなく，SPE0 で実行した場合と SPE1 で実行した場合とで違う振る舞いをするような単一のプログラムを作成することにします．

genrand_mine の呼び出し元の変更

実は，これまできちんと説明していませんでしたが，Hack the Cell 2009 の課題プログラムは，第 4 章で紹介した Hello World のように SPE 単体で処理が完結するような実装となっているため，PPE のプログラムが付属していません[*4]．したがって，Hack the Cell 2009 の課題プログラムを並列化するには，別途 `mt_kadai` (Hack the Cell 2009 の課題プログラムの SPE の実行ファイル) を呼び出す PPE のプログラムを作成するところから始めなければなりません．とはいえ，今回紹介する並列化では，PPE 側の処理は SPE コンテキストを 2 つ作成して，2 つの SPE 上で `mt_kadai` を同時実行し，その後メールボックスを通じて両 SPE の LS の先頭 EA を各 SPE に通知するくらいしかやることがないので，ここでは PPE のプログラムそのものの紹介は省略します．

また，SPE プログラムの `main` 関数にも若干手を加える必要があります．先ほど述べたように，SPE プログラムは，自身に振られた ID を元に何をすべきかを判断する (自身が SPE0 なのか SPE1 なのかを判断する) ので，それに必要な ID を PPE 側から受け取る必要があります．ID の通知は，SPE の `main` 関数の引数を利用するのが最も簡単なので，ここではそれを利用することにします．`main` 関数の引数で各 SPE に異なる ID を渡す方法については，第 8 章で紹介していますので，そちらを参照してください[*5]．また，各 SPE は，処理を始める前に相手の SPE の LS の先頭 EA をメールボックスから受け取る必要があります．これはどこでやってもよいのですが，ついでにこれも `main` 関数で行っておきましょう．このようにして取得した SPE の ID と

[*4] 第 4 章のコラムで紹介した elfspe2 を利用して動かすようになっています．

[*5] 第 4 章のコラムでも触れていますが，SPE の `main` の第 1 引数は，SPE を一意に識別することはできても，どちらが SPE0 でどちらが SPE1 なのかといったことを SPE 側で自主的に判断するような用途には使えません．

各 SPE の LS の EA は，グローバル変数として genrand_mine から参照することにします．別に main から順に関数の引数として渡していってもよいのですが，間の関数に変更を加えるのが手間なので，main 関数とそれを含むソースファイルの mt_kadai.c にリスト 11.1 に示す記述を追加し，genrand_mine の上の方にリスト 11.2 のように extern を追加して直接参照できるようにしておきます．

リスト 11.1 mt_kadai.c への変更

```
unsigned ls_high[2], ls_low[2]; // 各LSのEA保持用配列
int speid; // 自身のID保持用変数
...
int
main(unsigned long long spe, unsigned long long argp, unsigned long long envp)
{
    int i;
    speid = (int)argp; // IDを引数より取得

    // メールボックス経由でEAを受け取る
    for (i = 0; i < 2; i++) {
        ls_high[i] = spu_readch(SPU_RdInMbox);
        ls_low[i] = spu_readch(SPU_RdInMbox);
    }

    // 以下はオリジナルと同様
    compare();
    return 0;
}
```

リスト 11.2 SPE の ID と LS の EA を参照するための記述

```
extern unsigned ls_high[2], ls_low[2];
extern int speid;
```

ここで，リスト 11.1 で示したグローバル変数のうち，配列 ls_high が各 SPE の LS の EA の上位 32bit を保持し，配列 ls_low が各 SPE の LS の EA の下位 32bit を保持するものとします．また，ls_high と ls_low は，speid をインデックスとして参照することで，その ID に対応する SPE の LS の EA を取得できるものとします．これから述べる genrand_mine への変更や新規に追加する関数では，これらグローバル変数を参照して利用するものとして読み進めてください．

なお，課題のプログラムでは，チェックサムの答え合わせを genrand_mine の呼び出し元で行っていますが，今回の並列化では，チェックサムは Tempering 処理を行っている SPE1 の方で求まるので，genrand_mine の呼び出し元の方で speid を見て，SPE0 だったら答え合わせの処理を行わないようにしておく細工も必要になります．

genrand_mine への変更

これまで述べたように，ここでは genrand_mine の中の処理を SPE0 と SPE1 で手分けして行うようにしたことから，図 11.2 に示したような記述を genrand_mine に適用して，関数内で speid を元にした処理の切り替えを行うことにします．

まず，genrand_mine のメインループ (loop1 (num_rand を 624 で割った商) 回繰り返すループ) を，mt のアップデートを指定回数行う関数か，あるいは mt に Tempering 処理を指定回数適用する関数のどちらかを，speid の値に応じて呼び分ける記述に変更します．

次に，端数処理についてですが，こちらは要素数が 624 未満と少ないため，わざわざ並列化せずに，mt の更新も含めて，端数処理直前までの乱数の総和を持っている SPE1 で行わせることにします．

変更後のメインループと端数処理の部分をリスト 11.3 に示します．関数名や引数から想像がつくとは思いますが，ここで呼んでいる genrand_mine_update が，引数で指定した回数分の mt の更新を行う関数で，genrand_mine_tempering が，引数で指定した回数分の mt の全要素の Tempering 処理を行う関数になります．これらの中身については後ほど紹介します．また，端数処理は中身をほぼ全部省略していますが，ここには 6.5.1 で示したリスト 6.27 のループを二分した結果の後半の処理がそのまま入ります．

リスト 11.3　メインループの処理を speid ごとに切り替えて端数処理を SPE1 で行うように変更

```
    // メインループ
    if (speid == 0)
        genrand_mine_update(loop1);
    else
        vsum = genrand_mine_tempering(loop1);

    // 端数処理
    if (speid == 1) {
        vmt = (vec_uint4 *)vmt;
        ...
    }
```

また，転送先の EA を，DMA 転送を行うたびにいちいち計算するのは無駄なので，事前に計算をして変数に保持しておくことにします．SPE0 側では，SPE1 側の mt の EA と，mt の送信が完了したことを通知するための領域の EA が必要であり，SPE1 側では，SPE0 側に Tempering 処理が完了して新しい mt を受け取ることのできる状態となったという情報を通知するための領域の EA が必要となるので，これらを計算して初期化する処理を追加します．

それから，一方の SPE が変数の初期化処理を行っている間にもう一方の SPE が処理を開始してしまい，DMA 転送で初期化前の相手の変数を上書きしてしまったりすると処理が破綻するため，“プログラムがここまで到達したら相手が同じところに到達するまで待つ” という同期 (バリア同期) の仕組みも追加する必要があります．

このような変更をすべて盛り込んだ genrand_mine を，参照しているグローバル変数やマクロなどの宣言などとあわせて，リスト 11.4 に示します．

リスト 11.4　2 並列処理に対応させた genrand_mine

```
volatile int wp __attribute__((aligned(16)));
volatile int rp __attribute__((aligned(16)));
unsigned wp_ea_high, wp_ea_low;
unsigned rp_ea_high, rp_ea_low;
unsigned mt_ea_high, mt_ea_low;
#define TAGID 3
...
#define SET_EA(HIGH, LOW, EA)                     \
    HIGH = ((EA) >> 32);                          \
    LOW = (unsigned)(EA);

unsigned
genrand_mine(int num_rand, int speid)
{
    // 端数処理用の変数
    const vec_uchar16 vshuf = (vec_uchar16){4, 5, 6, 7, 8, 9, 10, 11,
                                            12, 13, 14, 15, 16, 17, 18, 19};
```

```
    ...
    int k, k1, k2, k3;

    int loop1 = num_rand / 624;
    int loop2 = num_rand - loop1 * 624;

    // DMAのタグにはTAGIDしか使わないのでここで設定
    spu_writech(MFC_WrTagMask, 1 << TAGID);

    // バリア変数の初期化
    barrier_init(speid);

    if (speid == 0) {
        unsigned long long ls_ea;
        rp = 1;
        wp = 1;
        // SPE1のLSのEA
        ls_ea = ((unsigned long long)ls_high[1] << 32) | ls_low[1];
        // SPE1のwpのEA
        SET_EA(wp_ea_high, wp_ea_low, ls_ea + (unsigned)&wp);
        // SPE1のmtのEA
        SET_EA(mt_ea_high, mt_ea_low, ls_ea + (unsigned)mt);
    } else {
        unsigned long long ls_ea;
        wp = 0;
        rp = 1;
        // SPE0のLSのEA
        ls_ea = ((unsigned long long)ls_high[0] << 32) | ls_low[0];
        // SPE0のrpのEA (どこまで読んだかを通知)
        SET_EA(rp_ea_high, rp_ea_low, ls_ea + (unsigned)&rp);
    }

    // 同期
    barrier(speid);

    // メインループ
    if (speid == 0)
        genrand_mine_update(loop1);
    else
        vsum = genrand_mine_tempering(loop1);

    // 端数処理
    if (speid == 1) {
        vmt = (vec_uint4 *)vmt;
        ...
    }

    // 最後に再度同期
    barrier(speid);

    return spu_extract(vsum, 0);
}
```

genrand_mine の外で宣言されている wp は，SPE0 が SPE1 に mt が更新されたことを通知するための領域で，rp は，SPE1 が SPE0 に Tempering 処理が済んだことを通知するための領域です．いずれも DMA 転送で通知を行うので，これらは 16Byte 境界にアラインさせて確保しています[*6]．また，その後の 6 個の変数は，相手の LS 上の変数の EA を保持するのに用います．

次の SET_EA は，64bit の EA を上位と下位に分けて代入するためのマクロです．大した機能があるわけでもありませんが，ソースが見にくくなるのを避けるために導入しています．

genrand_mine の中を見ていくと，最初にいきなり省略部分がありますが，ここには端数処理用の変数の宣言が入ります．中身は 6.3 のリスト 6.13 と同様になります．

[*6] 別にアラインさせなくてもこのプログラムは正常に動作します．元々4Byte の変数を単体で宣言した場合に，それが 4 で割り切れないアドレスから配置されることはなく，SPE0 と SPE1 でこれらの変数のアドレスは同一となるため，EA 空間にマップされた後もこれら変数の EA を 16 で割った余りは必ず等しくなるためです．

次に DMA 転送のタグマスクを設定しています．今回紹介する例では，タグは1種類しか使わないので，SPE0 と SPE1 の両方において `TAGID` をマスクにして設定しています．

その後で呼んでいる `barrier_init` は，バリア同期のためのデータを初期化する関数です．詳細は別途説明しますので，ここではひとまず読み飛ばしてください．

続いて，`speid` の値を見て `if` 文で処理を切り分けることを行っています．ここでは SPE0 と SPE1 で，それぞれ必要な変数の初期化を行っています．後で別途示しますが，今回の例では SPE1 の `wp` を 0 から1 にすることで SPE1 の LS の `mt` が DMA 転送で更新されたことを示し，SPE0 の `rp` を 0 から 1 にすることで SPE1 の Tempering 処理が完了したことを示すものとしています．SPE1 の `wp` は，SPE0 が DMA 転送で 1 を書き込み，SPE1 が値の検出直後に自ら 0 に戻します．SPE0 の `rp` も同様に，SPE1 が DMA 転送で 1 を書き込み，SPE0 が 0 に戻します．この処理に備えて，DMA 転送する値として，SPE0 では `wp` を，SPE1 では `rp` をそれぞれ 1 に初期化しています．また，SPE0 の `rp` と SPE1 の `wp` をともに 0 に初期化してしまうと，SPE0 が SPE1 から `mt` を書き込んでよいという指示が出るのを待ち続けてプログラムが一切動作しなくなってしまうため，SPE0 の `rp` は初期値を 1 として初期化しています．なお，この部分では，相手の LS の EA の計算も行っています．

次の `barrier` はバリア同期になります．中身については後で紹介しますが，自分の変数の初期化が終わったところでバリア同期を行うことで，“相手の初期化が終わるまで待つ” ことになり，先ほど述べたような，一方がどんどん先に進んでしまうことで他方が未初期化の変数を破壊するといったトラブルを未然に防ぐことができます．その後はリスト 11.3 に示した処理が行われ，SPE0 と SPE1 とで手分けして演算が行われます．そして最後にもう一度バリア同期を取って完了となります．

バリア同期の追加

先ほど説明を飛ばした `barrier` と `barrier_init` について説明します．バリア同期とは，待ち合わせ機構の一種で，乱暴な言い方をすれば “すべてのプロセッサで待ち合わせを行う” 仕組みとなります．`barrier_init` はバリア同期のための領域を初期化し，`barrier` は実際にバリア同期を行います．`barrier_init` と `barrier` の中身をリスト 11.5 に示します．今回の例では SPE0 と SPE1 の 2 プロセッサしかいないため，バリア同期は単純なフラグの読み書きで実装しています*7．`barrier` では，SPE1 がこの関数に差し掛かったら，SPE0 のバリア同期用の領域 (`barrier_buf`) に 1 を DMA 転送し，自身のバリア同期用の領域の値が 0 から 1 に変化するのをポーリングして待ちます．一方，SPE0 は，この関数に差し掛かると，自身のバリア同期用の領域が 1 になるのをポーリングして待ち，1 を検出したら SPE1 のバリア同期用の領域に 1 を DMA 転送で書き込みます．これにより，先にこの関数に到達した SPE は，もう一方の SPE がこの関数に到達するまで先に進まなくなります．`barrier_init` では，相手の SPE のバリア同期用の領域の EA を求め，自身のバリア同期用の領域に DMA 転送で書き込むのに使う 1 を設定しています．

*7 より多くの SPE に対応したバリア同期については，10.7.2 のリスト 10.20 を参照してください．やってることはこれと大差ありませんが ….

リスト 11.5 バリア同期関数

```
volatile unsigned barrier_buf[2][4] __attribute__((aligned(16))) = {{0}};
unsigned barrier_buf_eal, barrier_buf_eah;

static void
barrier_init(int speid)
{
    unsigned long long ls_base, barrier_buf_ea;
    if (speid == 0) {
        ls_base = ((unsigned long long)ls_high[1] << 32) | ls_low[1];
        barrier_buf_ea = ls_base + (unsigned)&barrier_buf[0][0];
        barrier_buf[0][0] = 1;
    } else {
        barrier_buf[1][0] = 1;
        ls_base = ((unsigned long long)ls_high[0] << 32) | ls_low[0];
        barrier_buf_ea = ls_base + (unsigned)&barrier_buf[1][0];
    }
    barrier_buf_eah = barrier_buf_ea >> 32;
    barrier_buf_eal = (unsigned)barrier_buf_ea;
}

static void
barrier(int speid)
{
    if (speid == 0) {
        // SPE1を待つ
        while (barrier_buf[1][0] == 0)
            ;
        barrier_buf[1][0] = 0;
        // SPE1を解放
        spu_mfcdma64(&barrier_buf[0][0], barrier_buf_eah, barrier_buf_eal,
                     16, TAGID, MFC_PUT_CMD);
    } else {
        // バリアに参加することをSPE0に伝える
        spu_mfcdma64(&barrier_buf[1][0], barrier_buf_eah, barrier_buf_eal,
                     16, TAGID, MFC_PUT_CMD);
        // SPE0からの解放通知を待つ
        while (barrier_buf[0][0] == 0)
            ;
        barrier_buf[0][0] = 0;
    }
}
```

バリア同期は，並列プログラミングにおいて非常に重要となる仕組みです．必ずこういったものを用意しておくべきでしょう．ただし，バリア同期を呼ぶと，その分処理効率は低下することになるので，無駄な呼び出しが発生しないようにする工夫も，同時に必要となります．

genrand_mine_update の導入

次に mt を更新する関数である genrand_mine_update について紹介します．中身をリスト 11.6 に示します．

リスト 11.6 genrand_mine_update

```
void
genrand_mine_update(int loop1)
{
    ...
    for (i = 0; i < loop1; i++) {

        k1 = 1;
        k2 = 99;
        k3 = 100;

        v1a[0] = vmt[0];
        ...
        v2b[3] = vmt[103];

        // 前半のデータが終わっているか確認（書き潰すので)
```

```
        spu_mfcstat(MFC_TAG_UPDATE_ALL);

        for (k = 4; k < 156; k += 4) {
            v1[0] = spu_shuffle(v1a[0], v1b[0], vshuf);
            ...
            v2b[3] = vmt[k3 + 3];

            if (k3 == 152)
                k3 = -4;
        }

        v1[0] = spu_shuffle(v1a[0], v1b[0], vshuf);
        ...
        vmt[155] = spu_xor(spu_xor(v2[3], spu_rlmask(vy[3], -1)), vmag01[3]);

        // 相手がまだ読み終わっていなかったら待つ
        while (__builtin_expect((rp == 0), 0))
            ;
        rp = 0;

        // データの転送要求
        spu_mfcdma64(vmt, mt_ea_high, mt_ea_low,
                     sizeof(vmt[0]) * 156, TAGID, MFC_PUT_CMD);

        // バッファを更新したことを相手に伝える
        spu_mfcdma64(&wp, wp_ea_high, wp_ea_low, sizeof(wp), TAGID,
                     MFC_PUTF_CMD);
    }
    return;
}
```

演算部分や変数の宣言の大部分が省略されていますが，この部分は 6.5.3 のリスト 6.32 に対して，6.6 のパイプラインの最適化と 6.7 のリスト 6.43 を適用した処理から，Tempering 処理 (`vy2` を用いた演算) を除いたものになります．足りない部分は適宜脳内で補完してください．

演算以外の部分に着目すると，演算処理の間に DMA 転送要求などがいくつか追加されていることがわかります．外側のループの最初で，内側のループで最初に処理を行うデータのロードを行った後に，`spu_mfcstat` を呼んで DMA 転送要求の処理完了をいきなり待っています．この部分では，外側のループの最後で行った SPE1 への最新の `mt` の DMA 転送が完了したかどうかを確認しています．これは，外側のループの 2 ループ目以後において，直前のループの最後で行った DMA 転送が終わる前に内側のループに突入してしまうと，DAM で転送中の `mt` をより新しい値で上書きしてしまう可能性があるためです．

`mt` の更新が済んだら，`rp` が 0 以外の値に変化するまで先に進まないようになっています．先ほど述べたように，`rp` は，SPE1 が "今ある `mt` の Tempering 処理が終わったから，次の `mt` で上書きしてもよい" ということを通知するための変数で，SPE1 による DMA 転送で値が 1 に書き換わることで通知が行われます．`rp` が 1 に変わったら，`mt` 全体を SPE1 の `mt` の領域に転送し，さらに `mt` が更新されたことを通知するために，SPE1 の `wp` に対して 1 を書き込んでいます (SPE0 の `wp` は 1 に初期化してある)．SPE1 の `wp` への DMA 転送は，コマンドが `MFC_PUTF_CMD` となっていることからもわかるように，フェンス属性のついた DMA 転送となっています．10.2 で述べたように，フェンス属性がついていると，先行して発行されている同じタグの DMA 転送よりも必ず後に処理されることが保証されるため，ここでは先行する `mt` の SPE1 への DMA 転送がすべて完了した後に，SPE1 の `wp` にフラグとして 1 が書き込まれることが保証されるようになります．これにより，SPE0 が `mt` を更新するよりも前に SPE1 が `mt` が更新されたと勘違いしてアクセスしてしまうことを防止しています．

なお，ここでは rp が 0 以外の値になるのを待つ while 文の中で，条件式に分岐ヒントをつけています．6.4 では特に触れませんでしたが，このように while 文の中にも分岐ヒントを入れることができます．

また，SPE1 の wp に 1 を書くと，その後，いつ SPE1 から rp に 1 が書き込まれてもおかしくない状況となってしまうため，これより後に rp を 0 に戻そうとすると，SPE1 からの rp への通知を取りこぼしてしまう可能性があります．そこで，ここでは，wp に 1 を DMA 転送で書き込むよりも前 (rp が 1 になったことを検出した直後) に，rp を 0 に戻しています．

genrand_mine_tempering の導入

続いて，mt の各要素の Tempering を行い，得られた乱数を足し込む処理を行う genrand_mine_tempering について紹介します．中身をリスト 11.7 に示します．

リスト 11.7 genrand_mine_tempering

```
vec_uint4
genrand_mine_tempering(int loop1)
{
    int i;
    vec_uint4 *vmt = (vec_uint4 *)mt;
    vec_uint4 vsum;
    vec_uint4 vy2[4];
    int k;

    vsum = spu_splats(0U);

    for (i = 0; i < loop1; i++) {
        // データが書かれるのを待つ
        while (__builtin_expect((wp == 0), 0))
            ;
        wp = 0;

        vy2[0] = vmt[0];
        ...
        vy2[3] = vmt[3];

        for (k = 4; k < 156; k += 4) {
            vy2[0] = spu_xor(vy2[0], spu_rlmask(vy2[0], -11));
            ...
            vy2[3] = vmt[k + 3];
        }

        // バッファの処理が完了したことを通知
        spu_mfcdma64(&rp, rp_ea_high, rp_ea_low, sizeof(rp), TAGID,
                     MFC_PUT_CMD);

        vy2[0] = spu_xor(vy2[0], spu_rlmask(vy2[0], -11));
        ...
        vsum = spu_add(vsum, vy2[3]);
    }

    return vsum;
}
```

こちらも演算部分が省略されていますが，6.5.3 のリスト 6.32 に 6.6 のパイプラインの最適化を適用したものをベースとしています．

演算以外の部分を見ると，まず，外側のループの最初で，wp の値が 0 以外になるまでループして待っていますが，この部分で SPE0 から DMA 転送によって mt の値が更新されるのを待っています．wp が 1 になったら，新しい mt の書き込みが完了したことを意味し，その後，SPE1 側から SPE0 の rp に対して 1 を書き込むまで mt が更新されないことが保証されるので，その

間に Tempering 処理を行います．なお，wp を即座に 0 に戻していますが，これは SPE1 の rp に 1 を DMA 転送で書き込むよりも前であれば，どこでやっても構いません．

Tempering 処理が完了する前でも，最後のロードが済んだ時点で mt の値は不要となるので，この例では最後の Tempering 処理の演算を行う前に，SPE0 の rp に対して DMA 転送で 1 を書き込んで，次の mt の値を転送してもよいことを通知しています．genrand_mine_tempering 内では，この DMA 転送の完了を spu_mfcstat で待つことをしていませんが，これは外側のループの先頭に戻り，wp が 0 以外の値になった時点で，SPE0 の rp への DMA 転送が完了したことが確実であると保証できるからです (SPE0 の rp に 1 が書き込まれない限り，SPE0 によって SPE1 の wp が更新されることはないため)．

11.1.4 並列化による性能改善

さて，長々と並列化について説明してきましたが，実際に並列化によって 6.7 で示した処理時間がどれだけ短縮するか見てみましょう．SPE1 側で計測した処理時間を表 11.1 に示します*8．

表 11.1 mt の更新と Tempering 処理の並列化による性能改善

状態	実行時間
並列化前	8,707,186
並列化後	6,254,095

処理時間が 30% ほど短縮しました．こんな単純な並列化でも十分に効果があるようです．

11.2 並列化後のプログラムのチューニング

次に，このプログラムをもう少し高速化できないか考えてみます．

11.2.1 チューニングの検討

genrand_mine_update と genrand_mine_tempering では，外側のループの中で wp と rp を利用して毎回同期を取りながら処理を進めているため，両者の 1 回のループにかかる時間はほぼ同一となります．しかしながら，両者で行っている処理の内容は大きく異なるため，どちらか一方で相手の処理が済むのを待つ時間が発生している可能性があります．ここではこの待ち時間に注目してチューニングができないかを探ってみます．

これら 2 つの関数で相手を待つ部分は，それぞれ while 文で wp もしくは rp の更新を待つ部分になります．実際に while 文がどれだけループしているのかを，リスト 11.8 のように数えてみると，genrand_mine_update ではループ回数がほとんど 0 であるのに対し，genrand_mine_tempering では 1 回 while 文に差し掛かるたびに 30 回以上もループしていることがわかりました．

*8 並列化後の処理には DMA 転送が含まれるため，処理時間が多少ぶれます．が，ここでは真面目に平均を求めたりせず，手を抜いて適当に 1 回測定した結果をそのまま使っています．

リスト 11.8 while 文にカウンタを入れて待ちの頻度を調べる

```
int n = 0;
...
    while (__builtin_expect((wp == 0), 0))
        n++;
```

このことから，genrand_mine_update で mt の更新が終わったときには，既に genrand_mine_tempering が前の mt の Tempering 処理を終えていて，SPE0 の rp に 1 が書かれている状態となっており，相手の処理が終わるのを待っているのは genrand_mine_tempering を行っている SPE1 の方だということがわかります．したがって，ここでは genrand_mine_update の方の処理をチューニングすることで，全体の処理時間を短縮できる可能性があることがわかります．

次に，genrand_mine_update の中で，各範囲にどれくらいの処理時間がかかっているのかを，リスト 11.9 のようにデクリメンタを利用して測定してみます．

リスト 11.9 デクリメンタで局所的な処理時間を調べる

```
unsigned long long dec = 0;
unsigned d1, d2;
...
    d1 = spu_readch(SPU_RdDec); // spu_mfcstat前の値取得

    // 前半のデータが終わっているか確認（書き潰すので）
    spu_mfcstat(MFC_TAG_UPDATE_ALL);

    d2 = spu_readch(SPU_RdDec); // spu_mfcstat後の値取得
    dec += (d1 - d2); // 差分を蓄積
```

結果を表 11.2 に示します．表より，genrand_mine_update の 1 回のループの処理時間のほぼ 1/4 を，spu_mfcstat でのストールが占めていることがわかります．

表 11.2 genrand_mine_update の処理時間の内訳

処理内容	実行時間
外側のループのロード	122,089
spu_mfcstat	1,577,775
内側のループ	4,603,520
外側のループの演算・ストア	178,172
DMA 転送要求など	204,245

このことから，genrand_mine_update と genrand_mine_tempering の処理のシーケンスは，図 11.3 のようになると推測されます．

11.2.2 データ転送のチューニング

mt の転送のダブルバッファリング化

spu_mfcstat でストールしている時間は，すなわち DMA 転送の完了を待つ時間になります．DMA 転送時間の隠蔽には，10.7.1 で紹介したダブルバッファリング化による DMA 転送のノンブロッキング化が効果的です．そこで，SPE0 から SPE1 への DMA 転送をダブルバッファリング化することを考えてみましょう．

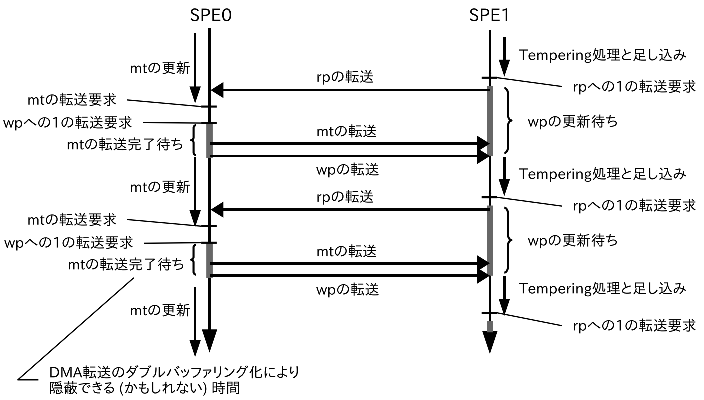

図 11.3　SPE0 と SPE1 の処理のシーケンス

ダブルバッファリング化するには，送信バッファと受信バッファをそれぞれ倍の量確保するのが簡単ですが，mt を倍のサイズ確保した場合の genrand_mine_update 関数の修正が面倒そうなので，SPE0 側では，mt のサイズを変えずに，代わりに mt の更新処理を前半と後半の 2 回に分けて行うことにします．すなわち，mt の後半部分の更新をしている間に前半部分の転送を行って，前半部分の更新をしている間に後半部分の転送を行うように変更を加えることで，ダブルバッファリング化します．ただし，その際，mt をちょうど真ん中で二分してしまうと，両側の要素数が 16 の倍数にならなくなってしまうため，端数処理が発生してしまいます．そこで，ここでは厳密に 2 等分するのではなく，mt の要素数が 16 で割り切れるところで，前半と後半がほぼ同じサイズになるように二分するものとします．具体的には前半を 302 要素，後半を 320 要素とします．

SPE1 の mt のリングバッファ化

一方，SPE1 側での Tempering をして結果を足し込む処理は，mt のバッファを倍に増やしたとしても，ループの先頭で vmt がどこを指しているのかを切り替えるだけで簡単に対応することができます．せっかくなので，ここでは SPE1 の mt のバッファサイズを 4 倍に増やしてリングバッファとしてみましょう．リングバッファの実装の例を図 11.4 に示します．

ここでは wp と rp を利用してリングバッファの管理を行っています．wp は “更新した mt を次に書く予定の場所” を示し，rp は “データが書かれたら最初に Tempering する場所” を示すものとします．よって両者が一致している状態は，“SPE1 の mt には Tempering 処理をしていないデータが存在しない” 状態となります．処理の流れから考えると，先に wp が増えて，それを追いかける形で rp が増えることになります．そのため，rp が wp に追いつくことは問題ありませんが，逆に wp が周回遅れの rp に追いついてしまうとリングバッファが空の場合と区別できなくなってしまうため，“追いつきそうになったら SPE0 側で rp が SPE1 によって DMA 転送で更新されるのを待つ” という処理が必要になります．

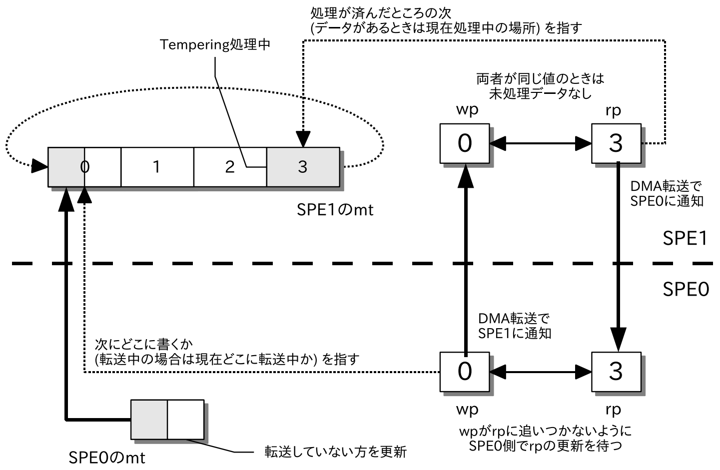

図 11.4　SPE1 側にリングバッファを導入した例

11.2.3　データ転送のチューニングのためのソースコードの変更

変数とその初期化処理の変更

ダブルバッファリング化やリングバッファ化を実現するためには，いくつかの変数やその初期化処理を変更する必要があります．リスト 11.10 に変数の宣言などの変更部分を示します．

リスト 11.10　リングバッファ導入時の変数の宣言の変更

```
#define BUFFERS 4
unsigned mt_ea_high[BUFFERS][2], mt_ea_low[BUFFERS][2];
...
static unsigned mt[N * BUFFERS] __attribute__((aligned(16)));
```

リスト 11.10 では，まず，これまで通常の変数だった `mt_ea_high` と `mt_ea_low` を 2 次元の配列にしています．これらは元々は SPE1 の `mt` の先頭だけを指していればよかったので配列にする必要はありませんでしたが，送信側のダブルバッファリング化によってデータを前半と後半の 2 回に分けて転送することになり，さらに受信側の受信領域を 4 倍のリングバッファとしたことで DMA 転送の対象となる領域が 8 つに増えることとなったため，このように配列としています．また，SPE1 のリングバッファ化のために `mt` のサイズも 4 倍にしてあります．これにより，SPE0 でも `mt` が 4 倍のサイズ確保されることになりますが，SPE0 では先頭の 1 つ分しか利用しません．

また，`genrand_mine` の中での変数の初期化処理も一部変更する必要があります．変更部分をリスト 11.11 に示します．

リスト 11.11　リングバッファ導入時の変数の初期化の変更

```
    if (speid == 0) {
        ...
        // SPE1のmtの各領域のEA
        ls_ea_mt = ls_ea + (unsigned)mt;
        for (i = 0; i < BUFFERS; i++) {
            // 前半の送り先のEAを導出
            SET_EA(mt_ea_high[i][0], mt_ea_low[i][0], ls_ea_mt);
            ls_ea_mt += sizeof(mt[0]) * 304;
            // 後半の送り先のEAを導出
```

```
            SET_EA(mt_ea_high[i][1], mt_ea_low[i][1], ls_ea_mt);
            ls_ea_mt += sizeof(mt[0]) * 320;
        }
    } else {
    ...
    wp = 0;
    rp = 0;
    ...
```

ここではSPE0の方のEAの初期化部分について，先ほど増やしたmt_ea_highとmt_ea_lowの各要素に対して適切なEAを設定するように変更しています．また，rpとwpの用途が変わるので，両方のSPEでこれらを0に初期化するように変更しています．

genrand_mine_update の変更

続いてgenrand_mine_updateをダブルバッファリング化します．genrand_mine_updateは以下の順で処理を記述することでダブルバッファリング化が可能です．

1. 前半を更新
2. 前のループの最後に発行した後半部分のDMA転送完了確認(後半を上書きして問題がないか)
3. 前半部分のDMA転送要求
4. 後半を更新
5. wpがrpに追いつかないか確認(追いつきそうならrpの更新を待つ)
6. 前半部分のDMA転送完了確認(次のループで前半を上書きして問題がないか)
7. 後半部分のDMA転送要求
8. SPE1のwpをDMA転送で更新

このようにしてダブルバッファリング化したgenrand_mine_updateのソースコードをリスト11.12に示します．例によって大量に省略されていますが，リスト11.6との大きな違いは，内側のループが2個に増え，前半でvmt[0]からvmt[75]までを処理し，後半で残りを処理している点になります．細かなループ内の処理はまったく同じなので，必要に応じて適宜補完してください．なお，wpのDMA転送要求にはフェンス属性がついているため，wpの値は同じタグで発行している前半部分のデータと後半部分のデータの両方のDMA転送が完了した後に書き込まれることが保証されます．

リスト 11.12　ダブルバッファリング化した genrand_mine_update

```
void
genrand_mine_update(int loop1)
{
    ...
    int wpnext;

    for (i = 0; i < loop1; i++) {
        k1 = 1;
        k2 = 99;
        k3 = 100;

        ...

        v2b[0] = vmt[100];
        ...

        for (k = 4; k < 76; k += 4) {
```

```
            ...
        }

        v1[0] = spu_shuffle(v1a[0], v1b[0], vshuf);
        ...
        vmt[75] = spu_xor(spu_xor(v2[3], spu_rlmask(vy[3], -1)), vmag01[3]);

        k1 = 77;
        k2 = 19;
        k3 = 20;

        v1a[0] = vmt[76];
        ...
        v2b[3] = vmt[23];

        // 前のループで発行した後半の転送が終わっているか確認
        // (これから書き潰すので)
        spu_mfcstat(MFC_TAG_UPDATE_ALL);

        // 前半のデータの転送要求
        spu_mfcdma64(&vmt[0], mt_ea_high[wp][0], mt_ea_low[wp][0],
                     sizeof(vmt[0]) * 76, TAGID, MFC_PUT_CMD);

        for (k = 80; k < 156; k += 4) {
            ...
        }

        v1[0] = spu_shuffle(v1a[0], v1b[0], vshuf);
        ...
        vmt[155] = spu_xor(spu_xor(v2[3], spu_rlmask(vy[3], -1)), vmag01[3]);

        // wpを更新した場合にrpに追いついてしまう場合はrpの更新を待つ
        wpnext = wp + 1;
        if (wpnext == BUFFERS)
            wpnext = 0;

        if (__builtin_expect((wpnext == rp), 0))
            while (wpnext == rp)
                ;

        // 前半の転送が終わっているか確認
        // (次のループの最初で書き潰すので)
        spu_mfcstat(MFC_TAG_UPDATE_ALL);

        // 後半のデータの転送要求
        spu_mfcdma64(&vmt[76], mt_ea_high[wp][1], mt_ea_low[wp][1],
                     sizeof(vmt[0]) * 80, TAGID, MFC_PUT_CMD);
        wp = wpnext;

        // 前後半の両方の転送が終わってからフラグを書く
        spu_mfcdma64(&wp, wp_ea_high, wp_ea_low, sizeof(wp), TAGID,
                     MFC_PUTF_CMD);
    }
    return;
}
```

genrand_mine_tempering の変更

最後に，genrand_mine_tempering をリングバッファ化します．リングバッファ化したものをリスト 11.13 に示します．こちらも大量に記述を省略していますが，基本は 11.7 と変わりません．

リスト 11.13　リングバッファに対応させた genrand_mine_tempering

```
vec_uint4
genrand_mine_tempering(int loop1)
{
    ...
    for (i = 0; i < loop1; i++) {

        // 両方同じだと未処理のデータがない状態
        if (__builtin_expect((rp == wp), 0))
```

```
            while (rp == wp)
                ;
        vmt = (vec_uint4 *)(mt + 624 * rp);

        vy2[0] = vmt[0];
        ...

        for (k = 4; k < 156; k += 4) {
            vy2[0] = spu_xor(vy2[0], spu_rlmask(vy2[0], -11));
            ...
            vy2[3] = vmt[k + 3];
        }

        vy2[0] = spu_xor(vy2[0], spu_rlmask(vy2[0], -11));
        ...
        vsum = spu_add(vsum, vy2[3]);

        // SPE0のrpを更新する
        rp++;
        if (rp == BUFFERS)
            rp = 0;
        spu_mfcdma64(&rp, rp_ea_high, rp_ea_low, sizeof(rp), TAGID,
                     MFC_PUTF_CMD);

    }
    return vsum;
}
```

リスト 11.7 と比べると，外側のループの最初で，rp が wp と違う値になるまで延々とループするようになっています．rp と wp が違う値となった場合に未処理の mt がリングバッファ上に存在することになるため，Tempering 処理を開始します．Tempering 処理が完了したら，SPE0 の rp を DMA 転送で進めることで，SPE0 にバッファが更新可能になったことを通知します．その際，DMA 転送要求にフェンス属性をつけておくことで，前のループで発行した rp への DMA 転送を追い抜いてしまうことを防いでいます．もし追い抜きが発生すると，SPE0 側の rp の値が実際のバッファの状態と異なったものとなってしまい，リングバッファの管理が破綻してしまいます．

11.2.4 データ転送のチューニングによる性能改善

以上でデータ転送のチューニングは完了です．実際にどれくらい性能が改善したか，実行時間を測定してみましょう．SPE1 側で計測した処理時間を表 11.3 に示します．

表 11.3 データ転送のチューニングによる性能改善

状態	実行時間
チューニング前	6,254,095
チューニング後	5,131,473

チューニングを行う前に比べて 2 割ほど処理時間が短縮されました．このようにして，DMA 転送の無駄な部分を切り詰めることで，大幅に処理時間を短縮できる場合があります．プログラム中で頻繁に DMA 転送を使用している場合は，最適化の 1 つの選択肢として十分検討する価値があると言えるでしょう．

第 12 章
プロブレムステート領域

> おれはだれの命令も受けぬ!! たとえ神の命令でもな!!
> — 武論尊・原哲夫『北斗の拳』より —

SPE は各種レジスタを EA 空間にマップして利用するための，MMIO と呼ばれる機能を備えています*1．これらレジスタ中で，ユーザが直接アクセスできると便利で，なおかつ直接アクセスした場合でもシステムの破綻につながらないような一部のものについては，OS などの特権ソフトウェアからだけでなく，ユーザレベル (プロブレムステート) で実行されているプログラムからも直接アクセスすることが許されています．このような領域をプロブレムステート領域と呼びます．

プロブレムステート領域も，MMIO でアクセス可能な領域の一部であることから，PPE からはロード・ストア命令による読み書きが可能です．プロブレムステート領域は，`spe_context_create` で SPE のコンテキストを生成する際に，フラグに `SPE_MAP_PS` をセットしておくことでユーザ空間にマップされ，ユーザプログラムから利用可能になります．第 9 章で紹介したメールボックスやシグナルにアクセスするための libspe2 の API や，10.6 で紹介した PPE から DMA 転送を要求するための libspe2 の API も，ごく一部の例外を除いて，内部では EA 空間にマップされたプロブレムステート領域のレジスタをアクセスしています*2．

本章では，このプロブレムステート領域を SPE から参照・利用する方法について紹介します．

12.1 プロブレムステート領域のレジスタのアドレス

プロブレムステート領域には様々なレジスタが用意されています．各レジスタは，アラインメントなどの関係から，物理アドレス (実アドレス) 上で飛び飛びのアドレスに配置されています．また，これらレジスタは機能ごとにグループ化されており，各グループは異なる物理ページに割り当てられています．これにより，OS などの特権ソフトがプロブレムステート領域のレジスタを機能ごとにユーザ空間へマップするかどうかを判断できるようになっています．このような理由から，プロブレムステート領域の EA は，機能グループ単位で取得しなければなりません．

---

*1 MMIO が一般的なメモリマップド I/O と何が違うのかさっぱりわからないのですが，Cell/B.E. 関連のドキュメントを見ると，特別な機能を指すような雰囲気で書かれているので，ここでもその表現にならいます．

*2 `spe_context_create` のフラグに `SPE_MAP_PS` をセットしなかった場合は，spufs のファイルの `read`・`write` を利用します．

12.1.1　プロブレムステート領域のレジスタのアドレスの取得

libspe2 は，プロブレムステート領域の各グループが EA 空間上でどのアドレスにマップされているのかを返す API を提供しています．

spe_ps_area_get：プロブレムステート領域のレジスタのグループごとの EA を取得

```
void *spe_ps_area_get(spe_context_ptr_t spe, enum ps_area area);
```

spe_ps_area_get は，spe で指定した SPE コンテキストの，area で指定したプロブレムステート領域のグループがマップされている EA 空間の先頭アドレスを返します．

area で指定できるプロブレムステート領域のグループと，その結果得られる EA から始まる各領域の型の一覧を表 12.1 に示します．

表 12.1　area で指定可能なグループ

グループ名	含まれるレジスタの概要・用途	得られる領域の型
SPE_MSSYNC_AREA	MFC マルチソース同期要求	spe_mssync_area_t
SPE_MFC_COMMAND_AREA	プロキシコマンド発行・結果確認	spe_mfc_command_area_t
SPE_CONTROL_AREA	メールボックスおよび制御用レジスタ	spe_spu_control_area_t
SPE_SIG_NOTIFY_1_AREA	シグナル 1 書き込み	spe_sig_notify_1_area_t
SPE_SIG_NOTIFY_2_AREA	シグナル 2 書き込み	spe_sig_notify_2_area_t

本関数の戻り値は void *型となっていますが，これはグループ間でのレジスタの配置が異なるためです．libspe2 のヘッダで各グループのレジスタの配置に合わせた構造体が定義されており，本関数の戻り値をそのグループの型にキャストすることで，容易に個々のレジスタの EA を得ることができます．各グループに対応した構造体の型と，その型に含まれるメンバを表 12.2 に示します．

本関数は，エラーの場合，NULL が返って errno にエラーの原因を指すコードがセットされます．

本関数を利用して，SPE_CONTROL_AREA に含まれる SPU_In_Mbox レジスタの EA を取得する例をリスト 12.1 に示します．この例では，最初に SPE_CONTROL_AREA の EA を得た上で，その中の構造体のメンバを参照することで，SPU_In_Mbox の EA を求めています．

リスト 12.1　spe_ps_area_get で SPE_CONTROL_AREA の SPU_In_Mbox レジスタの EA を得る例

```
    spe_spu_control_area_t *ctrl_area;
    // spethreads[0].contextのSPE_CONTROL_AREA領域の先頭EA取得
    ctrl_area = spe_ps_area_get(spethreads[0].context, SPE_CONTROL_AREA);
    // EAをキャストしてメンバ名で個別の要素を参照
    printf("EA of In_Mbox: %p\n", &ctrl_area->SPU_In_Mbox);
```

12.2　DMA 転送要求・完了確認

SPE_MFC_COMMAND_AREA 領域は，SPE の MFC に対するアクセスを行うためのレジスタを提供しています．SPE_MFC_COMMAND_AREA 領域に含まれるレジスタにアクセスすることで，SPE

表 12.2 各プロブレムステート領域の構造体とメンバ

領域の型	メンバ	メンバの概要
`spe_mssync_area_t`	`MFC_MSSync`	MFC マルチソース同期の要求
`spe_mfc_command_area_t`	`MFC_LSA`	`MFC_LSA` チャネルと同じ
	`MFC_EAH`	`MFC_EAH` チャネルと同じ
	`MFC_EAL`	`MFC_EAL` チャネルと同じ
	`MFC_Size_Tag`	転送サイズとタグの指定
	`MFC_ClassID_CMD`	`MFC_Cmd` チャネルとほぼ同じ
	`MFC_CMDStatus`	コマンドの書き込みの成否確認
	`MFC_QStatus`	コマンドキューの空き数の取得
	`Prxy_QueryType`	コマンド完了状況の調査方式設定・確認
	`Prxy_QueryMask`	タグマスク読み書き
	`Prxy_TagStatus`	タグマスクで指定したコマンドの完了状況取得
`spe_spu_control_area_t`	`SPU_Out_Mbox`	送信メールボックス読み出し
	`SPU_In_Mbox`	受信メールボックス書き込み
	`SPU_Mbox_Stat`	メールボックスのメッセージ数取得
	`SPU_RunCntl`	SPU の実行・停止の制御
	`SPU_Status`	SPU の各種ステータス取得
	`SPU_NPC`	SPU が次に実行する命令のアドレス
`spe_sig_notify_1_area_t`	`SPU_Sig_Notify_1`	シグナル 1 書き込み
`spe_sig_notify_2_area_t`	`SPU_Sig_Notify_2`	シグナル 2 書き込み

の外から MFC に対して DMA 転送要求を発行したり，その完了を確認したりすることができます．MFC に対して MMIO 経由で発行する DMA 転送要求などのコマンドをプロキシコマンドと呼びます．10.6 で紹介した PPE からの DMA 転送も，libspe2 の中でこの領域のレジスタをアクセスしてプロキシコマンドを発行することで実現しています．以下でこれらレジスタの使い方について紹介します．

12.2.1 DMA 転送要求で用いるレジスタ

最初に，MFC に対して SPE の外から DMA 転送要求を行う際に用いるレジスタの概要とフォーマットを示します．

MFC_LSA

`MFC_LSA` は，DMA 転送要求の際の LS 側のアドレスを指定するのに用いる 32bit 幅のレジスタです．32bit 幅分用意されていますが，LS の容量自体は 256KByte なため，Cell/B.E. では上位 14bit は無視されます．

MFC_EAH・MFC_EAL

`MFC_EAH` と `MFC_EAL` は，DMA 転送要求の際の EA 側のアドレスを指定するのに用いる 32bit 幅のレジスタです．名前から想像がつくかとは思いますが，`MFC_EAH` で EA の上位 32bit を指定し，`MFC_EAL` で下位 32bit を指定します．SPE から DMA 転送を要求する場合と同様に，`MFC_EAH` は何も指定しないとデフォルトで 0 が書かれているものとして扱われるようになって

いるため，PPE 側が 32bit バイナリの場合は，DMA 転送要求の際に MFC_EAH に値を書く必要はありません．また，これら 2 つのレジスタは実際には連続したアドレスに配置されており，MFC_EAH のアドレスを 64bit 幅でアクセスすることで，MFC_EAH と MFC_EAL の両方を同時に読み書きすることができるようになっています．

MFC_Size_Tag

MFC_Size_Tag は，DMA 転送要求の際の転送サイズとタグをまとめて指定するのに用いる 32bit 幅のレジスタです．実際には，内部は 16bit 幅の 2 領域に分かれており，図 12.1 に示すように，上位 16bit で転送サイズを指定し，下位 16bit でタグを指定します．なお，図の着色された部分は予約ビットとなっており，有効な値が設定できるのは上位 16bit の下位 15bit と，下位 16bit の下位 5bit の範囲に限られます．MFC に対して DMA 転送要求が可能なデータサイズが最大で 16KByte であり，タグとして利用可能な番号が 0 から 31 までということを考えると合点がいくでしょう．

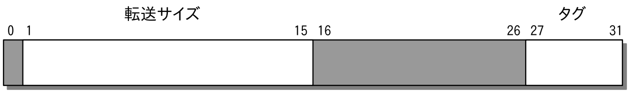

図 12.1　MFC_Size_Tag の構造

MFC_ClassID_Cmd・MFC_CMDStatus

MFC_ClassID_CMD は，DMA 転送要求の際に MFC にコマンドを書くのに用いる 32bit 幅のレジスタです．また MFC_CMDStatus は，コマンドの書き込みの結果を確認するのに用いる 32bit 幅のレジスタです．

これらは同じ実アドレスにマップされているため，EA も同じものとなります (構造体の中では共用体として表現されている)．したがって，MFC_ClassID_CMD から値を読んだ場合は MFC_CMDStatus から値を読んだのと同じ結果が返り，MFC_CMDStatus に値を書いた場合は MFC_ClassID_CMD に値を書いたのと同じことになります．

図 12.2 に MFC_ClassID_CMD の構造を，図 12.3 に MFC_CMDStatus の構造をそれぞれ示します．なお，MFC_ClassID_CMD の上位 16bit は tid と rid を指定するのに利用できるのですが，ここでは省略しています．MFC_ClassID_CMD の下位 8bit で指定可能なコマンドは，SPU がチャネルを介して MFC に対して DMA 転送要求を行う際に指定するコマンドと同一のもの (MFC_PUT_CMD や MFC_GET_CMD など) になります．ただし，10.6 でも述べたように，ここで要求できるのは通常の PUT 系と GET 系の DMA 転送のみ (フェンス，バリアおよびスタート属性の指定は可能) で，リスト DMA およびアトミック更新を要求することはできません．

図 12.2　MFC_ClassID_CMD の構造

図 12.3　MFC_CMDStatus の構造

MFC_CMDStatus から読み出せる 2bit のリターンコードの意味を表 12.3 に示します．両方のビットが 0 だった場合に，直前に MFC_ClassID_CMD に書いたコマンドが正しく受け付けられたことを意味します．

表 12.3　MFC_CMDStatus のリターンコード

ビット番号	意味
31	1 のときシーケンスエラーによってコマンドの書き込みに失敗したことを意味する
30	1 のときキューがいっぱいでコマンドの書き込みに失敗したことを意味する

なお，どういうわけか，MFC では MFC_CMDStatus を読み出さないと MFC_ClassID_CMD に書き込んだコマンドがキューに受け付けられないようになっているようです．設計としてどうかと思うのですが，MFC_ClassID_CMD への書き込みと MFC_CMDStatus からの読み出しは必ずペアで使うようにする必要があります．

MFC_QStatus

MFC_QStatus は，MFC のプロキシコマンドを受け付けるキュー (プロキシコマンドキュー) の現在の状態を読み出すことのできる 32bit 幅のレジスタです*3．図 12.4 に MFC_QStatus の構造を示します．

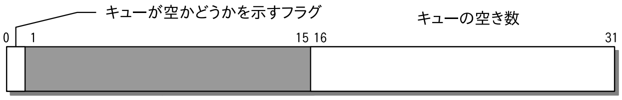

図 12.4　MFC_QStatus の構造

下位 16bit からは，MFC のプロキシコマンドキューの現在の空き数を取得することができます．また，最上位 1bit は，プロキシコマンドキューが空の場合に 1 となり，そうでない場合は 0 となります．

これまで紹介した DMA 転送要求のためのレジスタを用いて DMA 転送を行う場合，決められた順番でレジスタに値を書くことが (一応) 決められています．これを守らなかった場合，正しくパラメータが渡せなくなり，DMA 転送要求の発行に失敗する (もしくは不正なアドレスに DMA 転送を行う要求となり，プログラムがクラッシュする) 可能性があります．以下に DMA 転送要求を行う際のアクセスの順番を示します．

*3 10.6 でも触れましたが，プロキシコマンドキューは MFC が SPU からチャネルを介した DMA 転送要求を受け付けるキューとは別に用意されています．深さも SPU 用のキューが 16 段なのに対し，プロキシコマンドキューは 8 段となっています (Cell/B.E. の場合)．

1. MFC_LSA に LS のアドレスを書き込む
2. MFC_EAH に EA の上位 32bit を書き込む
3. MFC_EAL に EA の下位 32bit を書き込む (MFC_EAH とまとめて書き込むことも可)
4. MFC_Size_Tag に転送サイズとタグを書き込む
5. MFC_ClassID_CMD に DMA 転送のコマンドを書き込む
6. MFC_CMDStatus から DMA 転送要求のプロキシコマンドキューへの格納結果を読み出す (これにより DMA 転送要求が受け付けられる)
7. プロキシコマンドキューがいっぱいで受け付けられなかった場合は最初からやり直す

12.2.2　DMA 転送要求の完了確認で用いるレジスタ

MFC に対してプロキシコマンドを発行して DMA 転送要求を行った後は，プロブレムステート領域のレジスタを利用して完了状況の確認を行います．プロキシコマンドの完了状況の確認で用いるレジスタの概要とフォーマットを以下に示します．

Prxy_QueryMask・Prxy_TagStatus・Prxy_QueryType

Prxy_TagStatus，Prxy_QueryMask および Prxy_QueryType の 3 つのレジスタは，プロキシコマンドによる DMA 転送要求の完了状況を問い合わせる機能を提供するレジスタです．

Prxy_QueryMask は，SPU から DMA 転送要求の完了を確認する際に用いたタグマスクに相当する値を書き込むための 32bit 幅のレジスタです．タグに対応したビットに 1 を立てた値を設定しておくことで，後々DMA 転送要求の完了を問い合わせた際に，1 を立てておいたタグの DMA 転送についての完了状況を得ることができます．ビット番号とタグの対応はタグマスクと同様で，ビット 0 (MSB) がタグ 31 に対応し，ビット 31 (LSB) がタグ 0 に対応します．また，タグマスクと同様に，一度値を書いた後は値が勝手にクリアされることはないので，同じタグの DMA 転送要求の完了を繰り返し問い合わせるのであれば，毎回値を書き直す必要はありません．

Prxy_TagStatus は，Prxy_QueryMask で指定したタグのついた DMA 転送要求についての現在の完了状況を読み出すための 32bit 幅のレジスタです．読み出した結果は，Prxy_QueryMask と同様に，ビット 0 (MSB) がタグ 31 に対応し，ビット 31 (LSB) がタグ 0 に対応します．Prxy_QueryMask で確認を行うよう設定したビットが 0 だったら，そのビットに対応するタグの処理がまだ残っていることを意味し，1 だったら，そのタグの処理がすべて完了していることを意味します．

Prxy_QueryType は，DMA 転送要求の完了状況を PPE に割り込みで通知させる際に，どの条件が成立した際に割り込みを発生させるかを指定するのに用いる 32bit 幅のレジスタです．Prxy_QueryMask に設定したマスクで 1 となっているビットに対応したタグの DMA 転送要求について調査を行い，条件が満たされた時点で PPE へ割り込みを発生させます．割り込みは 9.2.5 にて紹介した PPE のイベント機能を利用して検出することができます．Prxy_QueryType の構造を図 12.5 に示します．

割り込みの条件は下位 2bit を利用して設定します．下位 2bit の値と条件を表 12.4 に示します．

また，このレジスタを読むことで，割り込み発生の有無を確認することができます．0 が読めた場合は，指定した条件が満たされたために割り込みが発生したことを意味し，それ以外の値が読めた場合は，まだ条件が満たされていないということを意味します．なお，ここまで説明して

図 12.5 Prxy_QueryType の構造

表 12.4 Prxy_QueryType で指定可能な条件

ビット	意味
00	条件指定なし (割り込みも発生しない)
01	Prxy_QueryMask で指定したタグのどれか 1 つに対応するタググループの全 DMA 転送の処理が完了したら割り込み
01	Prxy_QueryMask で指定したタグに対応するすべてタググループの全 DMA 転送の処理が完了したら割り込み
11	予約 (使用禁止)

から言うのもどうかと思うのですが，コンテキスト生成時に SPE_MAP_PS をフラグに与えてプロブレムステート領域をユーザプロセスの EA 空間にマップしていた場合，Prxy_QueryType に値を設定することで発生する SPE_EVENT_TAG_GROUP イベントは，イベントハンドラに登録できないようです．

12.3 メールボックスの利用

SPE_CONTROL_AREA 領域に含まれるレジスタのうち，SPU_Out_Mbox，SPU_In_Mbox および SPU_Mbox_Stat がメールボックスに関連するレジスタになります．これらレジスタに対してアクセスすることで，送信メールボックスからのメッセージの読み出し，受信メールボックスへのメッセージの書き込みおよび各種メールボックスに格納されているメッセージ数の取得を行うことができます．なお，割り込み送信メールボックスの値そのものを読み出すためのレジスタは，プロブレムステート領域には用意されていません[*4]．そのため，SPE からは割り込み送信メールボックスに書かれたメッセージを読み出せないことになります[*5]．

12.3.1 メールボックス関連のレジスタ

SPU_Out_Mbox

SPU_Out_Mbox は，SPE の送信メールボックスに書かれた値を読み出すための 32bit 幅のレジスタです．送信メールボックスは FIFO となっているため，一度読み出しを行うと，そのデータは送信メールボックスから消失します．MMIO に対するアクセスはブロックしないため，SPU が送信メールボックスに何も書いていない状態でこのレジスタを外部から読み出した場合，ブロックすることはありませんが，意味のある値が読み出されることもありません．そのため，

*4 割り込み送信メールボックスの値を読み出すためのレジスタは，OS などの特権ソフトウェアからのみアクセス可能な特権レジスタの一部として提供されています．

*5 PPE は spufs の提供するファイルから read することで値を読み出します．そのため，SPE_MAP_PS フラグをつけて SPE コンテキストを生成していた場合，送信メールボックスと割り込み送信メールボックスとでは読み出しにかかる時間に差が生じます．

PPEや複数のSPEからの読み出しが競合する可能性がある場合は，これらの間で排他制御を行った上でアクセスする必要があるでしょう．

SPU_In_Mbox

SPU_In_Mboxは，SPEの受信メールボックスに値を書き込むための32bit幅のレジスタです．受信メールボックスもFIFOであるため，SPUが値を読み出さないまま外部から次々と値を書いていくといっぱいになってしまいます(Cell/B.E.の場合は，4個でいっぱいになる)．受信メールボックスのFIFOがいっぱいの状態でさらに書き込みを行った場合，書き込んだ値は捨てられてしまう上に，書き込みに失敗したこともわからないことから，PPEや複数のSPEで同時にアクセスを行う場合は，これらの間で排他制御を行った上でアクセスする必要があるでしょう．

SPU_Mbox_Stat

SPU_Mbox_Statは，各メールボックスに格納されているメッセージ数を取得するための32bit幅のレジスタです．図12.6にSPU_Mbox_Statの構造を示します．図からもわかるように，単一のレジスタに送信メールボックス，受信メールボックスおよび割り込み送信メールボックスに格納されているメッセージ数の情報がひとまとめになっています．

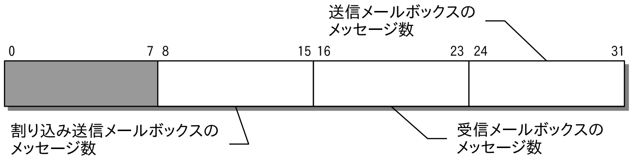

図12.6 SPU_Mbox_Statの構造

12.4 シグナルの利用

SPE_SIG_NOTIFY_1_AREAとSPE_SIG_NOTIFY_2_AREAは，シグナルを書き込むためのレジスタを提供する領域です．シグナル1とシグナル2に機能的な違いはありませんが，2つあるシグナルをユーザ用とカーネル用に使い分けられるようにするためか，これらは異なるグループに分かれて提供されています．なお，シグナルに書き込んだ値が上書きされるかORされるかは，プロブレムステート領域のレジスタからは設定することができません．

12.4.1 シグナル関連のレジスタ

SPU_Sig_Notify_1・SPU_Sig_Notify_2

SPU_Sig_Notify_1とSPU_Sig_Notify_2は，それぞれSPEのシグナル1およびシグナル2に対してシグナルを書き込むための32bit幅のレジスタです．

12.5　SPE から他の SPE の MFC に DMA 転送を要求する例

最後に，SPE からプロキシコマンドを発行して，他の SPE の MFC に DMA 転送要求を行う処理の例を紹介します．図 12.7 に，ここで紹介する例の処理の概要を示します．最初に，SPE0 (`speid` が 0 の SPE) が SPE1 (`speid` が 1 の SPE) の MFC に対して “PPE のバッファから SPE1 の LS 上のバッファにデータを読み出す DMA 転送要求” を DMA 転送で書き込みます (図の (1))．次に，これを受けた SPE1 の MFC が，PPE のバッファから SPE1 の LS 上にデータを読み出します (図の (2))．

続いて，SPE0 が SPE1 の MFC に対して，今度は “SPE1 の LS 上の別のバッファのデータを PPE のバッファに書き込む DMA 転送要求” を DMA 転送で書き込みます (図の (3))．そしてこれを受けた SPE1 の MFC が，SPE1 の LS 上のバッファから PPE のバッファへデータを書き込みます (図の (4))．

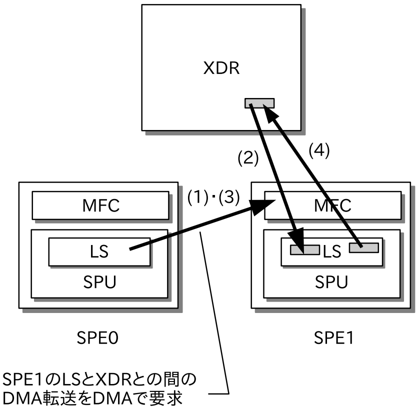

図 12.7　別の MFC に対して DMA 転送を要求する処理の流れ

なお，ここで紹介する例では，SPE から他の SPE のプロブレムステート領域のレジスタへのアクセスを API を使わずに実現しているため*6，これまでの例と比べると若干コードが入り組んだ構造となっています．また，使い方の例として記述しているため，処理の効率性は一切考慮していません．こんなこともできるんだという程度に見てください．

12.5.1　各レジスタのアドレスの管理

SPE から別の SPE にプロキシコマンドを発行するには，その別の SPE のプロブレムステート領域のレジスタの EA が必要となります．LS の EA と同様に，各 SPE のプロブレムステート領域のレジスタの EA は PPE 上でしか取得できないため，最初に PPE で libspe2 の API を用いてこれらの EA を取得し，それを SPE に伝える処理が必要となります．`spe_spu_control_area_t` などの構造体が SPE 側のヘッダでも定義されていれば楽なのですが，残念ながらそういった

*6 そもそも SPE からプロブレムステート領域のレジスタをアクセスするための API の存在を筆者は知りません．多分ないのではないかと ⋯．

ものは用意されていないようなので，ここではリスト 12.2 に示す，各領域の EA をまとめた context_ea_t 構造体を独自に定義し，これを用いて PPE と SPE の両方のプログラムで EA を共有することにします．PPE 側で，最初に libspe2 の API を利用して，このデータに情報を設定し，次にこのデータの先頭アドレスをメールボックスで SPE に通知して，最後に SPE から DMA 転送で読み出させるという算段です．

なお，復習となりますが，このような構造体を作る場合は，転送時にサイズが 16 の倍数となるようにしておかなければなりません．ここで示した context_ea_t は，メンバの数が偶数個なのでサイズがちょうど 16 の倍数となりますが，もしそうならない場合は構造体のメンバにパディングを加えるなどして 16 の倍数に揃えるようにしましょう．

リスト 12.2　context_ea_t 構造体の定義

```
typedef struct context_ea_t {
    unsigned long long lsa;
    unsigned long long eah;
    unsigned long long eal;
    unsigned long long size_tag;
    unsigned long long classid_cmd;
    unsigned long long cmdstatus;
    unsigned long long qstatus;
    unsigned long long querytype;
    unsigned long long querymask;
    unsigned long long tagstatus;
} context_ea_t;
```

12.5.2　PPE 側の処理

次に，PPE 側の main の処理をリスト 12.3 に示します．ppe_buf は，SPE1 から読み書きされるバッファとなります．また，ctx_ea は，リスト 12.2 で示した構造体の配列で，SPE のコンテキスト数分の要素を持ちます．

SPE のコンテキストを生成して別スレッドでコンテキストを実行する処理は，これまでとまったく同じなので省略しています．コンテキストを生成した後は，各 SPE のプロブレムステート領域の SPE_MFC_COMMAND_AREA のメンバの EA を取得して，ctx_ea の各要素のメンバにコピーしています．

ここまできたら，ppe_buf と ctx_ea の EA を，受信メールボックスを用いて SPE に伝えます．ここでは spe_in_mbox_write の機能を利用して 4 つのメッセージをまとめて送っています．各 SPE はこれを受けると動き始めます．後はメールボックスを利用して SPE と待ち合わせを行い，すべての SPE との待ち合わせが完了したら処理は終了となります．SPE が動き始めてから待ち合わせが完了するまでの間で，SPE 側でプロキシコマンドを用いた DMA 転送要求の発行や実際の DMA 転送などが行われます．したがって，待ち合わせが完了した時点で，ppe_buf の値は SPE1 からの DMA 転送によって更新されていることになり，printf でこれを出力して確認しています．

リスト 12.3 SPE から別の SPE に DMA 転送要求を行う際の PPE 側の処理

```
...
unsigned mbox_data[4];

volatile unsigned ppe_buf[4] __attribute__((aligned(16)));
context_ea_t ctx_ea[SPE_NUM] __attribute__((aligned(16)));
...
for (i = 0; i < SPE_NUM; i++) {
    // すべてのSPEのSPE_MFC_COMMAND_AREA領域のEAを配列の要素のメンバにセット
    spe_mfc_command_area_t *cmd;
    cmd = spe_ps_area_get(spethreads[i].context, SPE_MFC_COMMAND_AREA);
    ctx_ea[i].lsa = (unsigned long long)&cmd->MFC_LSA;
    ctx_ea[i].eah = (unsigned long long)&cmd->MFC_EAH;
    ctx_ea[i].eal = (unsigned long long)&cmd->MFC_EAL;
    ctx_ea[i].size_tag = (unsigned long long)&cmd->MFC_Size_Tag;
    ctx_ea[i].classid_cmd = (unsigned long long)&cmd->MFC_ClassID_CMD;
    ctx_ea[i].cmdstatus = (unsigned long long)&cmd->MFC_CMDStatus;
    ctx_ea[i].qstatus = (unsigned long long)&cmd->MFC_QStatus;
    ctx_ea[i].querytype = (unsigned long long)&cmd->Prxy_QueryType;
    ctx_ea[i].querymask = (unsigned long long)&cmd->Prxy_QueryMask;
    ctx_ea[i].tagstatus = (unsigned long long)&cmd->Prxy_TagStatus;
}
...
ppe_buf[0] = 0x0000cbea;

for (i = 0; i < SPE_NUM; i++) {
    // すべてのSPEにバッファのEAとctx_eaのEAを受信メールボックス経由で通知
    mbox_data[0] = (unsigned)(long)ppe_buf;
    mbox_data[1] = (unsigned)((long)ppe_buf >> 32);
    mbox_data[2] = (unsigned)(long)ctx_ea;
    mbox_data[3] = (unsigned)((long)ctx_ea >> 32);
    spe_in_mbox_write(spethreads[i].context, mbox_data, 4,
                      SPE_MBOX_ALL_BLOCKING);
}

// すべてのSPEと待ち合わせる
for (i = 0; i < SPE_NUM; i++)
    spe_out_intr_mbox_read(spethreads[i].context, mbox_data, 1,
                           SPE_MBOX_ALL_BLOCKING);

printf("PPE: ppe_buf[0]: 0x%08x\n", ppe_buf[0]);
```

12.5.3 SPE 側の処理

続いて SPE 側の処理を示します．SPE 側の処理では，DMA 転送で直接レジスタを叩くことを行うため，main 関数の他に，これを行うのに必要となる細々とした処理を行う関数をいくつか用意しています．main 関数について説明する前に，まず，これらを紹介します．

最初に，EA 空間上の任意の 4Byte の領域を DMA 転送で読み書きする関数である dma_read4 と dma_write4 の実装をリスト 12.4 に示します．これらはともに 4Byte の DMA 転送を行い，完了するまで待つ関数になります．dma_read4 が 4Byte の DMA 読み出しを行い，dma_write4 が 4Byte の DMA 書き込みを行います．SPE は，これらの関数を利用して，他の SPE の MFC のレジスタを読み書きします．これらの関数では，任意の 4 の倍数の EA が指定されても DAM 転送できるようにバッファを 16Byte 確保しており，DMA 転送を行う対象となる 4Byte の領域とアラインメントが揃う部分を用いて DMA 転送する記述としてあります．なお，これら関数は，DMA 転送要求を出した後，即座に完了を待つという非効率的な実装となっていますが，ここでは気にしないでください．効率性を追求するのであれば，完了を待つ処理は別関数に分けるなどすべきでしょう．

リスト 12.4 SPE から別の SPE に DMA 転送要求を行う際の SPE 側の処理 (1)

```
unsigned
dma_read4(unsigned long long ea)
{
    unsigned buf[4] __attribute__((aligned(16))); // アラインメント調整用
    unsigned idx = (ea & 0xf) >> 2;
    unsigned mask;

    mask = spu_readch(MFC_RdTagMask);
    spu_writech(MFC_WrTagMask, 1 << 31);
    // 指定のEAから4Byte読み出し
    spu_mfcdma64(&buf[idx], (unsigned)(ea >> 32), (unsigned)ea, 4, 31,
                 MFC_GET_CMD);
    spu_mfcstat(MFC_TAG_UPDATE_ALL);

    spu_writech(MFC_WrTagMask, mask);
    return buf[idx];
}

void
dma_write4(unsigned long long ea, unsigned data)
{
    unsigned buf[4] __attribute__((aligned(16))); // アラインメント調整用
    unsigned idx = (ea & 0xf) >> 2;
    unsigned mask;

    mask = spu_readch(MFC_RdTagMask);
    spu_writech(MFC_WrTagMask, 1 << 31);
    buf[idx] = data;
    // 指定のEAに4Byte書き込み
    spu_mfcdma64(&buf[idx], (unsigned)(ea >> 32), (unsigned)ea, 4, 31,
                 MFC_PUT_CMD);
    spu_mfcstat(MFC_TAG_UPDATE_ALL);
    spu_writech(MFC_WrTagMask, mask);
}
```

次に，リスト 12.4 で示した関数を用いてリモートの MFC に対して DMA 転送要求を発行する `spu_mfcdma64_remote` と，その完了を確認する `spu_mfcstat_remote` の実装を，リスト 12.5 に示します．`spu_mfcdma64_remote` は，引数 `rspeid` で指定される SPE のプロブレムステート領域にあるレジスタをアクセスし，プロキシコマンドとして DMA 転送要求を発行します．`spu_mfcdma64` と同様の感覚で他の SPE の MFC に DMA 転送要求を発行できるような API としてあります．

また，`spu_mfcstat_remote` は，引数 `rspeid` で指定される SPE に要求した引数 `tagid` のついた DMA 転送の完了を待ちます．中では，`Prxy_QueryMask` に，`tagid` に対応したタグのマスクを設定して，その後は `Prxy_TagStatus` の値の `tagid` に対応したビットが 1 になるまで繰り返し読み出しを行います．タグマスクの扱いなどが煩わしいので，こちらに関しては `spu_mfcstat` とは使い方を揃えていません．

最後に SPE の `main` 内での処理をリスト 12.6 に示します．`main` 関数では，まずバッファの初期化などを行っています．ここで SPE1 に関してのみ `buf2` を 0 以外の値に初期化させます．その後，割り込み送信メールボックスに値を書いて PPE と待ち合わせを行った後は，受信メールボックスから PPE 側のバッファの EA と `ctx_ea` の EA を取得し，後者から DMA 転送で `ctx_ea` の中身を LS 上に読み出します．

次に，SPE0 のみ `spu_mfcdma64_remote` を呼んで，SPE1 の MFC に対して PPE 側のバッファから `buf1` にデータを読み出す DMA 転送要求を発行します．ここで指定しているのが SPE0 の `buf1` のアドレスであることが気になるかもしれませんが，実行バイナリが共通で，さ

リスト 12.5 SPE から別の SPE に DMA 転送要求を行う際の SPE 側の処理 (2)

```
context_ea_t ctx_ea[SPE_NUM] __attribute__((aligned(16)));
...

void
spu_mfcdma64_remote(int rspeid, void *ls, unsigned eah, unsigned eal,
                    unsigned size, unsigned tagid, unsigned cmd)
{
    unsigned rd;
    do {
        // この順番で転送することが推奨されている
        dma_write4(ctx_ea[rspeid].lsa, (unsigned)ls);
        dma_write4(ctx_ea[rspeid].eah, eah);
        dma_write4(ctx_ea[rspeid].eal, eal);
        dma_write4(ctx_ea[rspeid].size_tag, (size << 16) | tagid);
        dma_write4(ctx_ea[rspeid].classid_cmd, cmd);
        // MFC_CMDStatusから読み出しを行わないと要求は受け付けられない
        rd = dma_read4(ctx_ea[rspeid].cmdstatus);
        // うまくいくまで繰り返す
    } while (rd & 0x3);
}

void
spu_mfcstat_remote(int rspeid, unsigned tagid)
{
    unsigned rd;
    // タグのマスクを設定
    dma_write4(ctx_ea[rspeid].querymask, 1 << tagid);
    do {
        // ひたすらPrxy_TagStatusを読み出してtagidに対応するビットが立つのを待つ
        rd = dma_read4(ctx_ea[rspeid].tagstatus);
    } while (rd != 1 << tagid);
}
```

らに buf1 はグローバル変数として確保してあるので，buf1 のアドレスはどの SPE でも同じになります (10.5.1 参照).

続いて，先に発行した DMA 転送の要求の完了を確認せずに，SPE1 の MFC に対して，buf2 から PPE のバッファへの DMA 転送の要求を発行しています．ここでは先に発行した DMA 転送要求と同じタグを用い，コマンドに MFC_PUT_CMD ではなく MFC_PUTF_CMD を指定しているため，SPE1 の MFC では，PPE から buf1 へのデータの読み出しが完了した後に buf2 から PPE へのデータの書き込みが行われることが保証されます．そして，これらの両方が終わったことを spu_mfcstat_remote で確認して，SPE0 のみが行う処理を抜けます.

最後に全 SPE で待ち合わせをした後，SPE1 の buf1 が更新されたかどうかを確認するために printf で出力を行っています.

この処理を実際に動かすと，以下のような結果が表示されます．PPE のバッファに SPE1 で設定した LS のバッファの値がコピーされており，一方で，SPE1 の LS のバッファの値も PPE から読み出したものに更新されていることがわかります.

```
% ./ps_dma_ppe
PPE: ppe_buf[0]: 0xce11b1e1
SPE1: buf1: 0x0000cbea
```

なお，今回示した例では省略していますが，PPE や複数の SPE から同時に特定の SPE の MFC に対して DMA 転送の要求が発行される可能性がある場合，DMA 転送要求を発行しようとしている PPE や SPE の間で必ず排他制御を行う必要があります．そうしないと，運悪く各種

リスト 12.6　SPE から別の SPE に DMA 転送要求を行う際の SPE 側の処理 (3)

```
unsigned buf1[4] __attribute__((aligned(16)));
unsigned buf2[4] __attribute__((aligned(16)));
...
    unsigned ppe_buf_eal, ppe_buf_eah, ctx_ea_eal, ctx_ea_eah;
    unsigned d;
    int speid = (int)argp;

    buf1[0] = 0;
    buf2[0] = speid == 1 ? 0xce11b1e1 : 0;
    d = speid;

    // とりあえず最初にPPEにmboxで通知して全 SPEが揃うのを待ってもらう
    spu_writech(SPU_WrOutIntrMbox, d);

    // EAを受信メールボックス経由で取得
    ppe_buf_eal = spu_readch(SPU_RdInMbox);
    ppe_buf_eah = spu_readch(SPU_RdInMbox);
    ctx_ea_eal = spu_readch(SPU_RdInMbox);
    ctx_ea_eah = spu_readch(SPU_RdInMbox);

    spu_writech(MFC_WrTagMask, 1 << TAGID);

    // LS上のctx_eaにPPE側で用意したctx_eaの値を読み出す
    spu_mfcdma64(ctx_ea, ctx_ea_eah, ctx_ea_eal, sizeof(ctx_ea),
                 TAGID, MFC_GET_CMD);
    spu_mfcstat(MFC_TAG_UPDATE_ALL);

    if (speid == 0) {
        // SPE0からSPE1のMFCに指示
        // PPEのバッファからSPE1のbuf1にデータを読み出す指示を出す
        spu_mfcdma64_remote(1, buf1, ppe_buf_eah, ppe_buf_eal,
                            sizeof(buf1), TAGID, MFC_GET_CMD);
        // PPEのバッファにSPE1のbuf2のデータを書き込む指示を出す
        spu_mfcdma64_remote(1, buf2, ppe_buf_eah, ppe_buf_eal,
                            sizeof(buf2), TAGID, MFC_PUTF_CMD);
        // 指示したDMA転送が終わるまで待つ
        spu_mfcstat_remote(1, TAGID);
    }

    spu_writech(SPU_WrOutIntrMbox, d);
    spu_readch(SPU_RdInMbox);

    if (speid == 1)
        // SPE1の場合buf1[0]を出力
        printf("SPE%d: buf1: 0x%08x\n", speid, buf1[0]);
```

レジスタへの書き込みが同時に発生した場合に，めちゃくちゃなパラメータの DMA 転送要求が発行されてしまうことになるためです．

第 13 章
外部関数呼び出し

それでも仙道なら… 仙道ならきっと何とかしてくれる…!!
— 井上雄彦『SLAM DUNK』より —

第 7 章などでも簡単に紹介しましたが，SPE の `printf` などの関数は，内部で PPE の libspe2 内に実装されている関数 (外部関数) を呼び出すように作られています．本章では，この外部関数呼び出しがどのように作られているのかについて解説します．また，外部関数をユーザが独自に追加する方法についてもあわせて紹介します．

13.1 stop-and-signal と外部関数呼び出し

SPE には “stop-and-signal” という名称の命令 (アセンブリ言語での表記は “`stop`”) が用意されており，この命令を呼ぶと，SPE は動作を停止します．その際，stop-and-signal 命令は，オペランドで指定された 14bit 幅の type と呼ばれる値をシグナルとして PPE に通知します．PPE のソフトウェアは，このシグナルの値を元に，SPE が停止した理由を判別することを行います．

stop-and-signal は，SPE プログラムが正常終了する際に呼ばれるようになっていますが，外部関数を呼び出す用途にも使われています．たとえば `printf` のような C 言語の標準関数 (C99) の一部や，`mkdir` などの POSIX.1 系の関数などは，SPE 上ですべてを実装するのが困難であるなどの理由から，実際に処理を行う部分は libspe2 側に実装されています．SPE のプログラムからこれらの関数を呼び出すと，関数の内部で stop-and-signal 命令が呼ばれ，SPE が動作を停止します．PPE は SPE の停止を検出することができるため，これを利用して後続の処理を PPE に委ねるのが，stop-and-signal を用いた外部関数呼び出しとなります．

libspe2 と SPE 向けの libc (newlib) の組み合わせで実現されている外部関数呼び出しは，以下の流れで処理されます．

1. SPE のプログラムで外部関数呼び出しにつながった関数を呼ぶ
2. 呼ばれた関数内で引数などの処理を行った後，stop-and-signal を呼ぶ
3. SPE が stop-and-signal で停止したことが PPE に通知される
4. PPE のプログラムが `spu_run` (`spe_context_run` の中で呼んでいるシステムコール) から戻る
5. PPE でシグナルの値から停止要因を調べ，対応する適切な関数を呼び出して処理を行う
6. SPE を stop-and-signal の次の次の命令から再開させる

13.2　標準で提供されている外部関数呼び出し

13.2.1　SPE 側の外部関数呼び出しの処理

以下では，`mkdir` などの，外部関数呼び出しを利用した SPE の関数を呼んだ際に，SPE がどのようにして停止し，どのようにして PPE に “すべきこと” を伝えるのかについて紹介します．

まず，SPE のプログラムで `mkdir` などの外部関数呼び出しで処理される関数を呼ぶと，その関数に渡された引数が LS 上の別の領域にコピーされます．次に，これから呼ぼうとしている stop-and-signal 命令の直後の 32bit 領域の下位 24bit に，引数の置いてある領域のアドレス (SPU から見たアドレス) を書き込みます．また，同じ領域の上位 8bit には，PPE に実行して欲しい処理に応じた ID (`mkdir` や `printf` などの関数ごとに用意された ID) を格納しておきます．

ここまでできたら stop-and-signal 命令を呼びます．その際，stop-and-signal 命令の type の値には `0x21xx` を指定します．“xx” の部分は，PPE 上で呼び出したい関数の属するグループに対応する値で，たとえば `mkdir` のような POSIX.1 系の関数を呼び出すのであれば `0x01` を指定し，`printf` などの C99 の標準関数を呼び出す場合は `0x00` を指定します．

stop-and-signal が呼ばれると，SPE は動作を停止し，ここで指定した type の値が PPE にシグナルとして通知されます．また，それと同時に，SPE が停止した際に次に実行する予定だった命令のアドレスも PPE に伝わるようになっています．PPE は，これら 2 つの情報を元に，以下のようにしてどの外部関数を実行すべきなのか (SPE がどの関数を呼び出したことにより動作を停止したのか) を判断します．

1. stop-and-signal によって通知される 14bit のシグナルの値の上位 6bit が `0x21` かどうかを見て，外部関数呼び出しのために停止したのかどうかを調べる
2. シグナルの値の下位 8bit を見て，どのグループに属する外部関数が呼ばれたのかを調べる (POSIX.1 系なのか C99 系なのかなど)
3. SPE が次に実行する予定であった命令のアドレスから 4Byte のデータを読み出す
4. 読み出した 4Byte の上位 8bit の値を見て，具体的にどの外部関数が呼ばれたのか (`mkdir` なのか `open` なのかなど) を調べる

PPE からは，SPE が停止した際に，後々SPE の実行を再開できるように SPE がこれから実行しようとしていた命令のアドレスを容易に取得できるようになっています．stop-and-signal を利用した外部関数呼び出しでは，この機構を利用して，stop-and-signal 命令の直後の 32bit に引数の置かれた領域のアドレスと処理の種類を示す ID を詰め込んで配置しておくことで，PPE に対して低コストで副次的な情報を渡すことを実現しています．なお，当然ですが，SPE で指定している外部関数のグループ番号や，そのグループ内での関数と ID の対応づけなどは，SPE (newlib) と PPE (libspe2) の間で統一されています．

PPE 側での外部関数が処理を完了した後は，SPE は stop-and-signal の次の次の命令，すなわち引数のアドレスなどが格納された領域の次の 32bit に格納された命令から処理を再開します．その際，外部関数の戻り値は，元々引数が置いてあった領域の最初の 4Byte に格納されてい

るので，必要に応じてこの値を読み出して関数の呼び出し元に返します．また，外部関数呼び出しでエラーが発生した際のエラーコード (`errno`) に関しては，引数の 4 番目の 4Byte の領域に入るものとされているので，必要に応じてこれを `errno` にコピーします．

これまでで述べた，stop-and-signal による外部関数呼び出しの際のデータ配置をまとめたものを図 13.1 に示します．

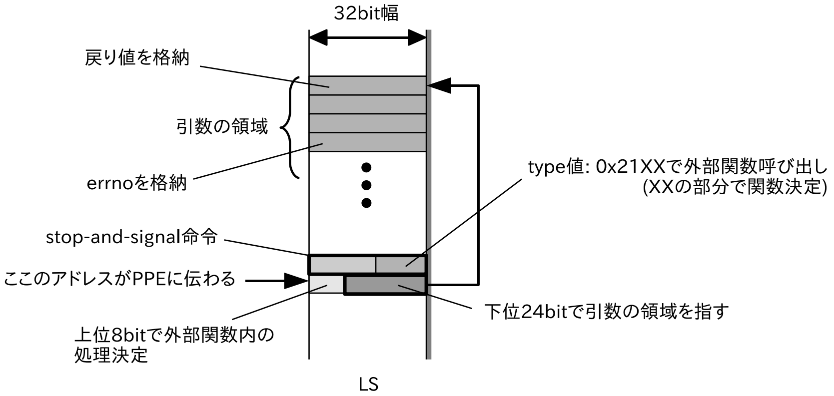

図 13.1　stop-and-signal を利用した外部関数呼び出し時のデータ配置

13.2.2　PPE 側の外部関数呼び出しの処理

前節でだいぶ述べてしまいましたが，次に PPE 側において，SPE が停止した際にどのようにして外部関数を実行するのかについて紹介します．

SPE が停止すると，PPE がそれを検出し，ユーザプログラムが libspe2 経由で呼び出していた `spu_run` システムコールから戻ります．その際，`spu_run` の戻り値 (32bit) の下位 8bit のどのビットが立っているかを見ることで，SPE がどのような理由で停止したのかがわかります．stop-and-signal 命令を呼んだために停止した場合，戻り値の LSB から 2bit 目 (`0x02`) が立ち，上位 16bit 中の下位 14bit に stop-and-signal 命令に渡したシグナルの値が入ります．よって，`spu_run` の戻り値を見ることで，SPE がどのグループの外部関数呼び出しで停止したのかを判別することができます．

どのグループの外部関数が呼ばれたかまで判断したら，次に実際にそのグループの処理を呼び出します (シグナルの下位 8bit で判断)．後で詳しく説明しますが，ユーザが独自に定義した関数を外部関数呼び出しとして追加している場合，ここからユーザの定義した処理に飛びます (ここまでは libspe2 の中の処理)．

グループの処理を呼び出す際に，引数として停止した SPE の LS の先頭アドレス (EA 空間上のアドレス) と，停止の原因となった stop-and-signal 命令の次の命令のアドレスが渡されます．グループごとの処理では，最初にこれら 2 つの引数を足し合わせて停止の原因となった stop-and-signal 命令の次の命令の EA 空間上でのアドレスを取得します．そして，ここから 4Byte を読み出し，LS 上の引数の置かれた領域のアドレスとグループ内の処理内容を指定する ID を得ます．後は ID ごとに処理を分岐させ，引数が必要な場合は LS の先頭アドレスに引数の

領域のアドレスを足し合わせて EA 空間上の引数の領域のアドレスを求め，そこから引数を読み出して利用します．戻り値を返す場合は，同様に引数の領域の最初の 4Byte に戻り値を書き込み，エラー時に `errno` を設定する必要がある場合は，引数の領域の 13Byte 目 (4 ワード目) からの 4Byte に `errno` の値を設定します．

グループごとの処理から `return` すると，libspe2 に処理が戻り，停止の原因となった stop-and-signal 命令の次の次の命令から処理を再開するように再度 `spu_run` を呼んで，SPE の処理を再開させることが行われます．

13.3　独自の外部関数の追加

次に，PPE プログラム上で独自に外部関数を追加する方法について紹介します．前節でも述べましたが，SPE が stop-and-signal で動作を停止した後は，libspe2 の中で停止原因の調査が自動で行われ，停止の原因が stop-and-signal 命令によるものであり，さらにその際のシグナルが `0x21xx` だった場合に，`xx` の部分に対応する外部関数が呼び出されます．この `xx` に対応する外部関数を PPE プログラムであらかじめコールバック関数として登録しておくことで，独自に定義した外部関数を SPE から呼び出せるようになります*1．

13.3.1　コールバック関数の作成

まず，実際に libspe2 の中から呼び出される外部関数となるコールバック関数を作成するところから始めてみます．コールバック関数は以下に示す型で作成します．

```
int function(void *ls_base, unsigned ls_addr);
```

`ls_base` は，stop-and-signal 命令で停止したコンテキストの LS の EA 空間上での開始アドレスを受け取るのに用います．また，`ls_addr` は，stop-and-signal の命令の次の命令 (直後の 32bit 領域) の SPU から見たアドレスを受け取るのに用います．

コールバック関数自体の戻り値は `int` 型になりますが，コールバック関数の戻り値は，SPE の外部関数呼び出しに対する戻り値ではない点に注意してください (SPE 自体に返す戻り値の設定方法については後述)．コールバック関数の戻り値は，libspe2 に，コールバック関数内の処理で異常な事態が発生していないかどうかを通知するのに用います．関数内の処理そのものが異常終了したのでなければ，必ず 0 を返さなければなりません．たとえばコールバック関数内でファイルを開こうとしたものの，パスが存在しなかったために開けなかったという場合でも，この関数は 0 を返す必要があります．0 以外の値を返した場合，libspe2 内のコールバック関数呼び出し部分は，コールバック関数そのものに深刻なエラーがあったと判断し，`spe_context_run` 関数を抜けてしまいます (SPE プログラムが処理を再開することなく終了してしまう)．

SPE からの引数を 1 つも取らず，戻り値も返すことのない関数に対応するコールバック関数は，リスト 13.1 のような記述となります．

*1 標準で搭載されている外部関数は，数が多いため，まずこのレベルでグループ分けをした上で，さらにグループ内で stop-and-signal 命令の直後の領域の上位 8bit を見て処理を選択していますが，独自に定義した外部関数がこの仕組みに従う必要はありません．

リスト 13.1　最もシンプルなコールバック関数の例

```
int
function(void *ls_base, unsigned ls_addr)
{
    return 0;
}
```

13.3.2　コールバック関数の登録

次に，コールバック関数を libspe2 に登録します．コールバック関数を登録するための関数は libspe2 によって提供されているので，ここではそれをそのまま利用します．コールバック関数はコンテキストごとに設定されるわけではなく，プログラム全体で共通のものが用いられます．

以下にコールバック関数の登録などに関連する libspe2 の API を示します．実際の使用例は後ほどあらためて紹介します．

spe_callback_handler_register：外部関数呼び出しのコールバック関数を登録

```
int spe_callback_handler_register(void *handler, unsigned callnum,
                                  unsigned mode);
```

spe_callback_handler_register は，libspe2 にコールバック関数を登録します．handler にコールバック関数の関数ポインタを渡し，callnum にコールバック関数の番号 (このコールバック関数がシグナルの下位 8bit (stop-and-signal で渡される 0x21xx の xx の部分) の値がいくつのときに呼び出されるか) を指定します．callnum は 8bit なので，0 から 255 までの値が指定可能ですが，0 から 3 までは標準で用意されている printf や mkdir などの外部関数呼び出しで用いられているため，ユーザは 4 以上の値を指定する必要があります．なお，Cell SDK 3.1 に付属している libspe2 では，どういうわけか 5 番も malloc_ea 系の外部関数で最初から使われてしまっています．番号は上の方から使う方が無難かもしれません．また，2 番と 3 番は，今のところ予約されているだけで，使われてはいないようです．

mode は，コールバック関数を新規に登録するか，あるいは既存の同じ番号のコールバック関数を置き換えるかを指定するのに用います．前者の場合は SPE_CALLBACK_NEW を指定し，後者の場合は SPE_CALLBACK_UPDATE を指定します．SPE_CALLBACK_UPDATE を指定した場合，callnum に 0 や 1 を指定することで，libspe2 の標準のコールバック関数を差し替えることも可能です．

登録に成功した場合は 0 が返り，失敗した場合は −1 が返って errno に原因が設定されます．

spe_callback_handler_deregister：外部関数呼び出しのコールバック関数の登録を解除

```
int spe_callback_handler_deregister(unsigned callnum);
```

spe_callback_handler_deregister は，callnum で指定した番号で登録されているコールバック関数の登録を解除します．ただし，libspe2 が予約している 0 から 3 までの番号のコールバック関数の登録は解除できません．

解除に成功した場合は 0 が返り，失敗した場合は −1 が返って errno に原因が設定されます．

spe_callback_handler_query：外部関数呼び出し用に登録されているコールバック関数を取得

```
void *spe_callback_handler_query(unsigned callnum);
```

spe_callback_handler_query は，現在登録されているコールバック関数を返します．関数を呼び出すと，callnum の番号で登録されているコールバック関数の関数ポインタが返ります．エラー発生時には NULL が返り，errno に原因が設定されます．

13.3.3 コールバック関数の呼び出しの許可

ユーザが独自に追加したコールバック関数は，コールバック関数を呼び出すことを許可された SPE コンテキストからしか呼び出せないようになっています．これを許可するには，リスト 13.2 のように spe_context_create でコンテキストを生成する際のフラグに対して SPE_EVENTS_ENABLE を OR します．

リスト 13.2　spe_context_create で SPE_EVENTS_ENABLE を指定する例

```
context = spe_context_create(SPE_MAP_PS | SPE_EVENTS_ENABLE, NULL);
```

ここまでで，PPE 側で必要なものは出揃いました．

13.3.4 SPE での外部関数の呼び出し

PPE 側のプログラムで独自に登録したコールバック関数は，SPE で__send_to_ppe という関数を呼ぶことで，簡単に外部関数として呼び出すことができます．

__send_to_ppe：外部関数を呼び出す

```
int __send_to_ppe(unsigned signalcode, unsigned opcode, void *data);
```

__send_to_ppe は，外部関数を呼び出します．元々この関数は，printf などの関数が呼ばれた際に newlib 内部で外部関数を呼び出すのに用いられている関数ですが，sys/syscall.h を include することで，ユーザプログラムからも直接利用することができます．

signalcode は，stop-and-signal でシグナルとして PPE に通知する値です．この関数を利用して，ある番号に登録してある外部関数を呼び出す場合は，signalcode に 0x2100 とその番号を OR した値を渡します．

また，opcode は，stop-and-signal の直後に置かれる 4Byte のデータの上位 8bit に格納される処理の ID で，data は同領域の下位 24bit に格納される引数や戻り値の受け渡しで用いる領域の開始アドレスになります．引数や戻り値のやりとりを行う必要がない場合でも，data には必ず 16Byte 以上の 4Byte 境界にアラインされた領域の先頭アドレスを渡す必要があります．これは，この関数が，内部で NULL のチェックなどを行うことなく data で指定された領域にアクセスするためです．opcode については，PPE 側で追加したコールバック関数の中で参照する実装となっていないのであれば，何を渡しても問題ありません．

PPE 側の外部関数で設定した戻り値は，この関数の戻り値として得られます．したがって，

`__send_to_ppe` を用いて外部関数呼び出しを行った場合，戻り値として 32bit よりも大きな値を受け取ることはできません．32bit よりも大きなサイズの戻り値を返すような外部関数を実装する場合は，外部関数側で戻り値を `data` で指定した領域の 2 ワード目や 3 ワード目もしくは 5 ワード目以降などに置くようにし，`__send_to_ppe` から戻った後に，SPE プログラム側で別途その領域を参照するなどの工夫が必要になります．

13.4 コールバック関数での引数の参照と戻り値の設定

先ほどは何もしないコールバック関数を示しました．以下では引数をきちんと受け取って，戻り値を正しく返すコールバック関数の実装方法を紹介します．

引数の参照

SPE で外部関数呼び出し時に指定した引数は，コールバック関数には直接渡されません．引数の値を得るには，まず，コールバック関数の第 1 引数 (外部関数を呼んだ SPE の LS の先頭 EA) と第 2 引数 (stop-and-signal の直後の領域の SPU から見たアドレス) を足し，stop-and-signal の直後の領域の EA を求めます．

次に，求めた EA からデータを 4Byte 読み出します．このデータの下位 24bit に，引数の置いてある領域のアドレス (EA ではなく SPU から見たアドレス) が格納されています．この値を再度第 1 引数の値と足すことで，引数が置かれている領域の EA を得ることができます．

リスト 13.3 にコールバック関数で SPE からの引数を参照する例を示します．なお，最初の stop-and-signal の直後のアドレスを求める部分において，`ls_addr` の LSB をマスクしていますが，これは SPE 上でイベントによる割り込み処理が有効になっている場合に，ここに 1 が立ってしまうためです．

以下の例では，単純に引数の領域の先頭から 4 ワード分を読み出して出力しています．なお，引数として渡された領域の中身の使い方には特に制限はなく，コールバック関数と，それを呼び出す SPE のプログラムの間で一貫性が維持されていれば自由に使うことができるのですが，libspe2 標準の外部関数呼び出しではここを 4Byte ずつに切って使っているようなので，ここでもそれにならっています．

リスト 13.3 コールバック関数で SPE の引数を参照する例

```
int
function(void *ls_base, unsigned ls_addr)
{
    // stop-and-signal命令の直後の領域の値を取得 (ls_addrのLSBはクリアしておく)
    unsigned arg = *((int *)(ls_base + (ls_addr & ~1)));
    int *args;

    // 取得した値をLS上のアドレスとして，再度LSを参照するとそこに引数がある
    args = (int *)(ls_base + (arg & 0x00ffffff));

    for (i = 0; i < 4; i++)
        printf("args[%d] = 0x%08x\n", i, args[i]);
    ...

    return 0;
}
```

SPE に対する戻り値

次に，コールバック関数から SPE に対して戻り値を渡す方法を紹介します．SPE に対する戻り値は，コールバック関数の中で LS 上の引数の領域の先頭 4Byte に書き込むことで渡します．また，外部関数呼び出しでエラーが発生した際に，原因を SPE に通知するために errno も伝えたい場合は，引数の領域の 13Byte 目 (4 ワード目) からの領域にエラーコードを書き込みます．

先ほどのコールバック関数に戻り値の設定を書き加えたものをリスト 13.4 に示します．この例では戻り値として 1 を返しています．また，errno に 0 を設定していますが，ここで 0 を入れておくと，SPE 側で既存の errno 値を上書きすることを防ぐことができます．errno と関係のない関数呼び出しでも，ここには毎回 0 を書くようにしましょう．

リスト 13.4　コールバック関数から SPE に戻り値を返す例

```
int
function(void *ls_base, unsigned ls_addr)
{
    unsigned arg = *((int *)(ls_base + (ls_addr & ~1)));
    int *args;

    args = (int *)(ls_base + (arg & 0x00ffffff));

    for (i = 0; i < 4; i++)
        printf("args[%d] = 0x%08x\n", i, args[i]);

    args[0] = 1; // 戻り値
    args[3] = 0; // errno

    return 0;
}
```

13.5　コールバック関数の追加と呼び出しの具体例

これまでのおさらいとして，実際に PPE 側で独自のコールバック関数を外部関数として登録し，それを SPE から呼び出す処理の例を紹介します．

13.5.1　PPE 側の実装

最初に PPE 側の関数をリスト 13.5 に示します．

test_handler がコールバック関数です．中では先ほどの例と同じように LS 上の引数が置かれた領域の EA を求め，そこから 4 個の引数を読み出して出力し，1 つ目に戻り値として 1 を書き，4 つ目に errno の更新がないことを示す 0 を書いています．

一方，main の方では，最初に spe_callback_handler_query を全外部関数の番号に対して適用し，現在コールバック関数が設定されている番号がどれかを調べています*2．

次に，test_handler をコールバック関数として登録しています．その際，対応する番号には 0xff を設定しています．よって，このコールバック関数を呼ぶには，SPE 側で stop-and-signal で 0x21ff をシグナルとして送ればよいことになります．

続いて，SPE のプログラムをロードし，spe_context_create で，第 1 引数に SPE_EVENTS_

*2 単なる API の使い方の一例で，特別な意味のある処理ではありません．

リスト 13.5 自前のコールバック関数とその登録の例

```
int
test_handler(void *ls_base, unsigned ls_addr)
{
    unsigned arg = *((int *)(ls_base + (ls_addr & ~1)));
    int i, *args;

    args = (int *)(ls_base + (arg & 0x00ffffff));

    for (i = 0; i < 4; i++)
        printf("args[%d] = 0x%08x\n", i, args[i]);

    args[0] = 1;
    args[3] = 0;

    return 0;
}

int
main(int argc, char *argv[])
{
    int i;
    spe_program_handle_t *program;
    unsigned entry = SPE_DEFAULT_ENTRY;
    spe_stop_info_t stop_info;
    spe_context_ptr_t context;

    // コールバック関数が設定されている番号を調べてみる
    for (i = 0; i < 256; i++)
        if (spe_callback_handler_query(i))
            printf("callback %d is already used\n", i);

    // コールバック関数登録
    spe_callback_handler_register(test_handler, 0xff, SPE_CALLBACK_NEW);

    program = spe_image_open("stop_and_signal_spe");

    // ここでSPE_EVENTS_ENABLEを追加しないと自前のコールバック関数を呼べない
    context = spe_context_create(SPE_MAP_PS | SPE_EVENTS_ENABLE, NULL);

    spe_program_load(context, program);
    spe_context_run(context, &entry, 0, NULL, NULL, &stop_info);

    // SPEのプログラムが終わったらコールバック関数の登録を解除
    spe_callback_handler_deregister(0xff);

    spe_context_destroy(context);
    spe_image_close(program);
    return 0;
}
```

ENABLE を OR して SPE コンテキストを生成しています．これにより，このコンテキストにおいて，ここで追加した外部関数呼び出しが使用可能となります．

この後は，普通にコンテキストを実行し，コンテキストの処理が終了したら，一応 spe_callback_handler_deregister を呼んでコールバック関数の登録を解除しています．

13.5.2 SPE 側の実装

一方，これを呼び出す SPE のプログラムはリスト 13.6 のようになります．

最初に sys/syscall.h を include しています．これをやらないと__send_to_ppe を呼ぶところでコンパイル時に警告が出ます．また，errno を参照するために errno.h も include しています．

main の中で errno に EAGAIN を設定していますが，これは外部関数を呼び出した後で，errno

リスト 13.6　SPE から外部関数呼び出しで自前のコールバック関数を呼び出す例

```
#include <stdio.h>
#include <spu_intrinsics.h>
#include <sys/syscall.h>
#include <errno.h>

int
main(unsigned long long spe, unsigned long long argp, unsigned long long envp)
{
    unsigned args[4];
    int ret;

    errno = EAGAIN;

    args[0] = 0xce11b0e0;
    args[1] = 0x01234567;
    args[2] = 0x89abcdef;
    args[3] = 0xcbeacbea;

    ret = __send_to_ppe(0x21ff, 0x10, args);

    printf("ret = %d\n", ret);
    printf("errno = %d\n", errno);

    return 0;
}
```

が更新されていないことを確認するために適当に初期化しているだけで，右辺の値にはまったく意味がありません．

その後，引数として利用する配列 `args` に適当な値を代入した上で，`__send_to_ppe` を呼んでいます．`__send_to_ppe` の第 1 引数には，先ほど PPE 側で登録したコールバック関数を呼び出すために `0x21ff` を渡し，第 3 引数には引数を置いた領域のアドレスを渡しています．第 2 引数で `0x10` を渡していますが，この値自体には特別な意味はありません．PPE のコールバック関数で，stop-and-signal の直後の領域から読み出した 32bit の上位 8bit をきちんと捨てているかどうかを確認するために渡しています．

13.5.3　実行結果

先ほど実装した PPE のプログラムを実行すると，以下のように表示されるはずです．実行結果より，外部関数の 0 番，1 番および 5 番が使用中であることが確認できます．また SPE から外部関数として登録したコールバック関数がきちんと呼び出されており，戻り値も正しく戻っていることがわかります．`errno` も PPE のコールバック関数で 0 を指定したため，元の値のまま上書きされていません．

```
%  ./stop_and_signal_ppe
callback 0 is already used
callback 1 is already used
callback 5 is already used
args[0] = 0xce11b0e0
args[1] = 0x01234567
args[2] = 0x89abcdef
args[3] = 0xcbeacbea
ret = 1
errno = 11
```

13.6 __send_to_ppe の内部処理と stop-and-signal 命令の生成

最後に，__send_to_ppe 関数の中でどのようにして stop-and-signal 命令を呼んでいるのかを簡単に紹介します．

stop-and-signal は，オペランドとして，レジスタではなく即値を取る命令です．何を言っているのかというと，命令の設計上，stop-and-signal で通知するシグナルの値 (type) にはレジスタに設定したものをそのまま使うことができないのです．シグナルの値は，命令に即値として値そのものを埋め込まなければなりません．したがって，stop-and-signal 命令で任意のシグナル PPE に伝えられるようにするには，動的に命令を LS 上に作って，そこにプログラムカウンタを飛ばすことになります．

一体どうやっているのかと思われるかもしれませんが，newlib の__send_to_ppe の実装はリスト 13.7 のようになっています (見やすくするために一部修正).

リスト 13.7　newlib の stop-and-signal で任意のシグナル値を発行する実装

```
int
__send_to_ppe(unsigned signalcode, unsigned opcode, void *data)
{

    int newerrno;
    unsigned combined = ((opcode << 24) & 0xff000000)
                      | ((unsigned)data & 0x00ffffff);

    unsigned stopfunc = {
        signalcode,  // この部分がstop-and-signalとなる
        (unsigned)combined, // 引数のアドレスなど
        0x4020007f, // nop
        0x35000000  // bi $0 (returnに相当)
    };

    void (*f)(void) = (void *)&stopfunc;
    si_sync(); // これをやらないと動かない
    f(); // スタック上に確保した命令列に飛ぶ
    newerrno = ((unsigned *)data)[3];

    if (newerrno)
        errno = newerrno;

    return ((unsigned *)data)[0];
}
```

最初に，combined という変数に，第 2 引数と第 3 引数を混ぜた値を代入しています．次に，stopfunc という配列を確保しています．1 要素目に第 1 引数の signalcode が入っていますが，実はこの配列はデータのふりをした命令列で，後々ここにプログラムカウンタを飛ばして中身を命令列とみなして実行することを行います．combined が 2 要素目に格納されていることから想像がつくかもしれませんが，1 要素目が stop-and-signal になります．第 1 引数をそのまま実行しようとしているのに，なぜこれが stop-and-signal 命令として解釈されるのか不思議に思われるかもしれません．実は，stop-and-signal のフォーマットは，上位 18bit がすべて 0 [*3]で，下位 14bit の値でシグナルを指定するようになっています．よって，ここでは第 1 引数が 0x3fff 以

[*3] 厳密には上位 18bit のうちの下位 7bit は予約ビットらしいのですが，現在のところ 0 埋めしておけば問題ないようです．

下であれば，必ず stop-and-signal として解釈される命令が作られることになります．

この部分にプログラムカウンタを飛ばすために，この配列の先頭を，引数も戻り値も持たない関数の関数ポインタに見立てて関数ポインタ f に代入し，その後，関数として f を呼ぶことで，ここに飛んでいます．これにより，関数の引数に応じた任意のシグナルを発行できる stop-and-signal の呼び出しを実現しています．

なお，f を呼ぶ直前に "si_sync" という見慣れない型固有組み込み関数が呼ばれていますが，これを呼ばないと，この処理は正常に動きません．si_sync は，ここまでに発行されたストア処理をすべて終わらせてから，その後の命令を読み込んで実行することを保証させる "sync" 命令に対応した関数です．これを呼ばなかった場合，分岐ヒント命令によって stopfunc へ分岐するヒントが与えられていると，signalcode や combined の値が stopfunc に書き込まれる前に stopfunc の中身のプリフェッチが行われてしまい，結果として誤ったシグナルを送る stop-and-signal が呼ばれることになってしまいます．

第 14 章 イベント

> あ … ありのまま　今起こったことを話すぜ！
> 「おれは奴の前で階段を登っていたと思ったら　いつのまにか降りていた」
> — 荒木飛呂彦『ジョジョの奇妙な冒険』より —

6.9 や 10.4.8 で少しだけ触れましたが，SPE にはイベントと呼ばれる機構が存在し，割り込みを利用したイベント処理にも対応しています．SPE らしいことをさせようと思うとあまり出番はないのかもしれませんが，そもそも知名度が低いせいで使われていないだけなのではないかという気がしなくもないので，本章で簡単に紹介しておきます．

14.1　イベントの概要と管理

SPE のイベントは全部で 12 種類あり，デクリメンタのカウンタが 1 周したときや，メールボックスやシグナルへの書き込みが生じたとき，アトミック更新のリザベーションが消失したときなどに発生します．SPE のユーザプログラムは，これらのイベントを，チャネルを介して管理します．表 14.1 にイベントを管理する際に用いるチャネルを示します．

表 14.1　イベントの管理で用いるチャネル

チャネル名	R/W	深さ	ブロック	用途
SPU_RdEventStat	R	1	する	発生中のイベントの読み出し
SPU_WrEventMask	W	1	しない	イベントマスクの設定
SPU_RdEventMask	R	1	しない	設定中のイベントマスクの読み出し
SPU_WrEventAck	W	1	しない	イベントへのアクノリッジの発行

SPE でのイベントは 2 段階で管理されています．まず，SPE 内部のハードウェアがイベントの発生を検出すると，内部のペンディングイベントレジスタと呼ばれるレジスタの，発生したイベントに対応するビットに 1 が立ちます．このレジスタの各ビットはラッチとなっており，一度イベントが発生して 1 が立つと，ソフトウェア的にリセットされるまでずっと 1 を保持したままとなります．SPE のプログラムは，ペンディングイベントレジスタで 1 が立っているイベントのうち，イベントマスクと呼ばれるビットマスクで 1 を立てているイベントについてのみ，イベントの発生状況を検出することができます．イベントマスクは `SPU_WrEventMask` チャネルに値を書き込むことで設定することができます．

イベントマスクで検出対象としているイベントの発生状況は，`SPU_RdEventStat` チャネル

から取得することができます．SPU_RdEventStat チャネルはブロッキング型のチャネルなため，イベントが何も発生していない状態で読みにいくと SPU はストールし，イベントが新規に発生するまで返ってこなくなります．イベントマスクで検出対象としているイベントが新規に発生するなどすると，SPU_RdEventStat チャネルから値が読み出せるようになります．SPU_RdEventStat チャネルから読み出せる値は，イベントマスクとペンディングイベントレジスタの値の AND を取ったものとなります．したがって，SPU_RdEventStat チャネルを読み出すと，イベントマスクで 1 を立てているイベントの，その時点での発生状況を一括して取得できることになります．

ペンディングイベントレジスタのビットはラッチとなっているため，一度イベントが発生すると，イベントの要因が解除されてもイベントに対応するビットは 1 の状態を維持し続けることになります．したがって，これを放置していると，別のイベントが発生した際に，既に処理済みのイベントも新規に発生したものとして誤検出されてしまうことになります．ペンディングイベントレジスタのビットを下げるには，SPU_WrEventAck チャネルに対して，解除するイベントのビットに 1 を立てたビットマスクを書き込みます．通常は，SPU_RdEventStat チャネルから読み出した値をそのまま書き込めばよいでしょう．

SPU_RdEventStat チャネルは，以下の条件のいずれかが成立した場合に値が格納され，読み出せるようになります．

- イベントマスクで有効となっているイベントが新規に発生した (ペンディングイベントレジスタのビットが 0 から 1 になった) 場合
- SPU_WrEventMask チャネルに書き込みを行ってイベントマスクを設定したら，イベントマスクで有効となっているイベントが既に発生していた (ペンディングイベントレジスタのビットが 1 だった) 場合
- SPU_WrEventAck チャネルに書き込みを行ってイベントを解除したが，イベントマスクで有効となっているイベントのうち，ペンディングイベントレジスタで 1 になっているものが残った場合

14.2　イベントの種類

SPE で発生する 12 種類のイベントの中で，本書の内容と深く関係していそうなものを以下で紹介します．括弧内は各イベントに対応したフラグになります．フラグは，イベントマスクの設定など，チャネルを介したイベントの管理で用います．

- MFC のコマンドキュー空き発生 (MFC_COMMAND_QUEUE_AVAILABLE_EVENT)

  これまでいっぱいだった MFC のコマンドのキューのコマンドが処理されて空きができた (MFC_Cmd チャネルのチャネルカウンタの値が 0 から変化した) 際に発生します．

- MFC タググループステータス更新 (MFC_TAG_STATUS_UPDATE_EVENT)

  MFC_WrTagUpdate チャネルに対して書き込みを行った後，MFC_RdTagStat チャネルから

コマンドの完了状況を読み出せるようになった (MFC_RdTagStat チャネルのチャネルカウンタが 0 から 1 になった) 際に発生します.

- リスト DMA ストール発生 (MFC_LIST_STALL_NOTIFY_EVENT)
  実行中のリスト DMA において，notify メンバに 1 が設定されている要素の処理を終えた (MFC_RdListStallStat チャネルのチャネルカウンタの値が 0 から 1 になった) 際に発生します.

- リザベーション消失 (MFC_LLR_LOST_EVENT)
  アトミック更新の getllar で取得したリザベーションが，他のプロセッサによるメモリ書き込みによって消失した際に発生します.

- デクリメンタ 1 周 (MFC_DECREMENTER_EVENT)
  32bit あるデクリメンタの MSB (ビット 0) の値が 0 から 1 に遷移した，すなわち 0x00000000 からさらに 1 減って 0xffffffff になった際に発生します.

- 受信メールボックス未処理データ発生 (MFC_IN_MBOX_AVAILABLE_EVENT)
  これまで空だった受信メールボックスに値が書き込まれた (SPU_RdInMbox チャネルのチャネルカウンタの値が 0 から変化した) 際に発生します.

- 送信メールボックス空き発生 (MFC_OUT_MBOX_AVAILABLE_EVENT)
  これまでいっぱいだった送信メールボックスから値が読み出され，空きができた (SPU_WrOutMbox チャネルのチャネルカウンタが 0 から変化した) 際に発生します.

- 割り込み送信メールボックス空き発生 (MFC_OUT_INTR_MBOX_AVAILABLE_EVENT)
  これまでいっぱいだった割り込み送信メールボックスから値が読み出され，空きができた (SPU_WrOutIntrMbox チャネルのチャネルカウンタが 0 から変化した) 際に発生します.

- シグナル 1 未処理データ発生 (MFC_SIGNAL_NOTIFY_1_EVENT)
  今まで何も書かれていなかったシグナル 1 に新規に値が書かれた (SPU_RdSigNotify1 チャネルのチャネルカウンタが 0 から 1 に遷移した) 際に発生します.

- シグナル 2 未処理データ発生 (MFC_SIGNAL_NOTIFY_2_EVENT)
  今まで何も書かれていなかったシグナル 2 に新規に値が書かれた (SPU_RdSigNotify2 チャネルのチャネルカウンタが 0 から 1 に遷移した) 際に発生します.

ここで紹介した以外のイベントや，上記イベントに関するより詳細な情報を知りたい場合は，文献 [5] などを参照してください.

14.3 イベントの検出と処理

イベントの検出方法には，ループなどでチャネルやチャネルカウンタを定期的にアクセスすることで発生を検出する同期式イベント検出と，割り込みを用いて検出を行う非同期式イベント検出があります.

14.3.1 チャネルアクセスによる同期式のイベント検出

チャネルアクセスによる同期式のイベント検出は，定期的に SPU_RdEventStat チャネルのチャネルカウンタを読むなどして行います．SPU_RdEventStat チャネルのチャネルカウンタの値が 1 だったら，イベントが発生しているということになるため，実際に SPU_RdEventStat チャネルを読み出してイベントの処理を行います．なお，イベントの発生を検出した場合は，まず SPU_RdEventStat チャネルを読んでどのイベントが発生したかを調べ，次に SPU_WrEventAck チャネルに対して，発生を検出した全イベントに対するアクノリッジを書いて，ペンディングイベントレジスタのビットを下げることを行い，その上でイベントごとの処理を行うことが推奨されています．

14.3.2 専用命令による同期式のイベント検出

SPE には，イベントの発生の有無をチェックして，発生していた場合に指定のアドレスに分岐するイベントチェック用の命令として，bisled，bisledd，bislede の 3 つの命令が用意されています．これを利用することで，イベント関連のチャネルを直接読む場合に比べて効率的にイベントの検出を行うことができます．これらの命令は，型総称組み込み関数として呼び出すことができます[*1]．以下にフォーマットを示します．

```
spu_bisled(fn);
spu_bisled_d(fn);
spu_bisled_e(fn);
```

いずれの命令も，引数に void (* fn)() 型の関数 (引数がなく戻り値が void 型の関数) を取り，この関数が呼ばれた時点でイベントが発生していたら (SPU_RdEventStat チャネルのチャネルカウンタが 0 でなかったら) fn に飛ぶような分岐処理を行うことになります．spu_bisled は，単に fn に飛ぶだけなのですが，spu_bisled_d は，その際に割り込みをオフにします．また，spu_bisled_e は，逆に fn に飛ぶ際に割り込みをオンにします．この命令を使うことで，非常に低いコストでイベント発生のチェックが可能となるため，定期的にイベントの発生を確認したい場合におすすめです．また，割り込みのオンオフが同時にできるので，割り込みでのイベント処理と併用するのにも有効そうです．

専用命令を用いた同期式のイベント検出においても，イベント検出後は，14.3.1 の最後で述べた方法と同じ手順でイベントの処理を開始する必要があります．

14.3.3 割り込みによる非同期式のイベント検出

これまで紹介した 2 つのイベントの検出方法は，どちらもプログラム中で定期的にイベントの発生を確認することでイベントの検出を行うものでしたが，SPE は，これ以外に割り込みを用いることでイベントを検出する方法も備えています．SPE への割り込みは，SPU_RdEventStat チャネルのチャネルカウンタの値が 0 から 1 になったタイミングで発生します．割り込みが発生

[*1] その他の一般的な分岐命令は，型総称組み込み関数や型固有組み込み関数を用いて明示的に呼び出すことはできません．

すると，現在実行中のアドレスが SRR0 というレジスタに退避され，プログラムは 0 番地へ飛びます．したがって，0 番地からの領域に適切な割り込み処理を行うルーチンを用意しておくことで，イベントの発生をいちいちポーリングすることなく検出して処理することが可能になります．

Cell SDK では，First-Level Interrupt Handler (FLIH) と呼ばれる割り込みハンドラが最初から用意されています．この割り込みハンドラは，0 番地に割り付けておくことで，割り込み発生時にチャネルの読み書きやレジスタのスタックへの退避などの必要な処理を一通り行った後，内部のテーブルで保持しているイベントごとの処理ルーチン (Second-Level Interrupt Handler (SLIH)) を呼び出します．ユーザは，自前のイベント処理ルーチンを SLIH として登録しておくことで，割り込み発生時に自前の処理を行うことが可能となります．

イベントの非同期処理を伴わないプログラムを `spu-gcc` でコンパイルしても，0 番地からの領域に FLIH がリンクされることはありませんが，プログラム中で `spu_timer.h` というヘッダファイルを `include` し，`spu_slih_register` という関数を用いて SLIH を登録する処理を記述することで，それに依存する FLIH がコンパイル時に 0 番地からの領域にリンクされ，イベント発生時に実行されるようになります．`spu_slih_register` の使い方については後述します．

FLIH の処理

FLIH は，最初に 80 本の汎用レジスタの内容をスタックに退避した後，`SPU_RdEventStat` チャネルを読んで，現在どのイベントが発生しているのかを調べます．次に，読み出した値をそのまま `SPU_WrEventAck` チャネルに書き込んで，先ほど発生を確認したすべてのイベントについて，ペンディングイベントレジスタのビットを下げます．その後，現在設定されているイベントマスクを読み出して，そこから先ほど発生を確認したイベントのフラグを下げたものを新たなイベントマスクとして設定し直します．

ここまで来たら，SLIH の呼び出し処理に入ります．FLIH は，発生を確認したイベントのうち，フラグのビットが MSB 寄りにあるものから順番に，対応する SLIH を呼び出します．後で詳しく述べますが，発生を確認したイベントは，イベントマスクと同じフォーマットのビットマスクの形で SLIH に引数として渡されます．SLIH は，引数として受け取ったビットマスクから，その中で処理したイベントに該当するビットをクリアして戻り値として返します．FLIH は，SLIH の戻り値を 1bit ずつ上位から見て，まだ 1 のビットが残っていることを確認したら，そのビットに対応する SLIH を呼び出します．これを戻り値が 0 になるまで繰り返します．

その後，元々設定されていたイベントマスクの値を復元し，スタックに積んだ値をレジスタに戻した上で，最後に割り込みからの復帰命令を呼びます．割り込みからの復帰命令が呼ばれると，SPU は SRR0 に保持しているアドレスに復帰します．

割り込みの有効化・無効化

SPE では割り込みはデフォルトで無効になっています．ユーザプログラム中でこれを有効にするには，`spu_ienable` という型総称組み込み関数を呼びます．また，逆に割り込みを無効化させる場合は，`spu_idisable` という型総称組み込み関数を呼びます．これらの関数は引数を取らず，戻り値もありません．

spu_slih_register：SLIH を登録する

```
void spu_slih_register(unsigned event_mask, unsigned (*slih)(unsigned));
```

spu_slih_register は，FLIH にイベントごとの SLIH を登録するのに用いる関数です．引数として，SLIH として登録する関数と，その関数で処理するイベントのビットを立てたビットマスクを取ります．上記フォーマットでは，event_mask がイベントを示すビットマスクで，slih がイベントごとの SLIH として登録する関数となります．slih の型からもわかるように，イベントハンドラは unsigned 型の引数を 1 つ取り，unsigned 型の戻り値を返す関数となります．先ほど述べたように，このとき引数として渡されるのが，FLIH で検出して SLIH でまだ処理が済んでいないビットの立ったビットマスクになり，SLIH が戻り値として返すことが期待されるのが，そのビットマスクから SLIH 内で処理したビットを下げた値となります．

14.3.4 割り込みによるイベント処理の例

ここで，実際に spu_slih_register を用いて自前で用意した SLIH を登録し，FLIH を通じてイベント処理を行う例を紹介します．まず，SLIH の例をリスト 14.1 に示します．

リスト 14.1　デクリメンタの 1 周とシグナル書き込みに対する SLIH の例

```
#include <spu_timer.h> // SLIHを使う場合にincludeする
...
volatile unsigned dec_loop = 0; // デクリメンタのループ数を数えるカウンタ

// デクリメンタ1周時のSLIH
unsigned
decrementer_handler(unsigned status)
{
    // 何回1周したかを数えておく
    dec_loop++;
    // statusからフラグを下げる (下げないと再度この関数が呼ばれる)
    return status & ~MFC_DECREMENTER_EVENT;
}

// シグナル検出時のSLIH
unsigned
signal_handler(unsigned status)
{
    unsigned d;

    if (status & MFC_SIGNAL_NOTIFY_1_EVENT) {
        // シグナル1を検出したら読み出してstatusからフラグを下げる
        d = spu_readch(SPU_RdSigNotify1);
        printf("signal notify #1: 0x%08x\n", d);
        status &= ~MFC_SIGNAL_NOTIFY_1_EVENT;
    }

    if (status & MFC_SIGNAL_NOTIFY_2_EVENT) {
        // シグナル2を検出したら読み出してstatusからフラグを下げる
        d = spu_readch(SPU_RdSigNotify2);
        printf("signal notify #2: 0x%08x\n", d);
        status &= ~MFC_SIGNAL_NOTIFY_2_EVENT;
    }
    return status;
}
```

decrementer_handler は，デクリメンタが 1 周した際に呼ばれる SLIH で，signal_handler は，シグナル 1 もしくはシグナル 2 に何らかの値が新規にセットされた場合に呼ばれる SLIH になります．status で 1 が立っているイベントが，今回の割り込みの原因となったイベントの中

で，その時点で SLIH によって処理されずに残っているイベントになります．その時点で発生しているイベントの状態を示しているわけではない点に注意してください．

デクリメンタが 1 周した場合でも，イベントを再度発生させるために何らかのチャネルを読み出したりする必要はないため，`decrementer_handler` では単純に何回デクリメンタが 1 周したのかをカウントするカウンタの値を 1 増やした後，`status` から `MFC_DECREMENTER_EVENT` のフラグを下げた値を `return` しています．FLIH は，ここで `return` された値を元に，次にどの SLIH を呼ぶかを判定するので，必ずきちんとビットを下げた値を返すようにしましょう (ビットを下げないと無限ループに陥る)．

一方，シグナルに関しては，値をチャネルから読み出して抜き取っておかないと，次にシグナルに何か書き込まれてもイベントが発生しない状態となってしまうため，`signal_handler` では `status` にフラグが立っていた場合に，該当するシグナルのチャネルから読み出しを行っています．なお，リスト 14.1 の例では，読み出しを行ったレジスタについて個別に `status` のフラグを下げていますが，無条件に `MFC_SIGNAL_NOTIFY_1_EVENT` と `MFC_SIGNAL_NOTIFY_2_EVENT` の両方を下げるような記述としてもよいでしょう．

これら SLIH を，`spu_slih_register` を介して FLIH に登録する際の例をリスト 14.2 に示します．最初の `spu_slih_register` で，`MFC_DECREMENTER_EVENT` 発生時に `decrementer_handler` が呼ばれるように登録を行っています．2 つ目の `spu_slih_register` では，`MFC_SIGNAL_NOTIFY_1_EVENT` と `MFC_SIGNAL_NOTIFY_2_EVENT` のどちらが発生した場合でも `singal_handler` が呼ばれるように登録を行っています．`spu_slih_register` の第 1 引数はイベントマスクと同じフォーマットとなっているため，このように複数のイベントに対して一括で同一の SLIH を登録することができます．

リスト 14.2 spu_slih_register で SLIH を FLIH に登録する例

```
// デクリメンタ1周とシグナル受信の検出のSLIHを登録
spu_slih_register(MFC_DECREMENTER_EVENT, decrementer_handler);
spu_slih_register(MFC_SIGNAL_NOTIFY_1_EVENT | MFC_SIGNAL_NOTIFY_2_EVENT,
                  signal_handler);

// デクリメンタ1周とシグナル受信のみを検出するようイベントマスクを設定
spu_writech(SPU_WrEventMask, MFC_DECREMENTER_EVENT
                           | MFC_SIGNAL_NOTIFY_1_EVENT
                           | MFC_SIGNAL_NOTIFY_2_EVENT);

// デクリメンタ初期化
spu_writech(SPU_WrDec, 0xffffffffU);

// イベントによる割り込みを有効化
spu_ienable();
```

その後，`SPU_WrEventMask` チャネルを介して，これら 3 つのイベントを SPE から検知できるようにするためのイベントマスクを設定し，最後に `spu_ienable` を呼んで割り込みを有効化しています．

14.3.5 ファントムイベント

ペンディングイベントレジスタのビットがクリアされていないためか，SPE がプログラムの実行を開始した直後からイベントが発生していることがまれにあります (マスクを書き込んだら即

座に SPU_RdEventStat チャネルが読み出し可能になる状態)．これはファントムイベントと呼ばれ，文献 [5] などにも記載があります．ファントムイベントでは，たとえばシグナル 1 のイベントが発生しているのに SPU_RdSigNotify1 のチャネルカウンタの値は 0 といった状況も生じ得るようなので，割り込みマスク設定後にイベントの発生状況を確認し，ファントムイベントらしきものがあったら SPU_WrEventAck チャネルへの書き込みを行って，あらかじめイベントを解除しておくのが安全です．

これが正しい処理なのかいまいち確信が持てませんが，先に示したプログラムであれば，spu_ienable を呼ぶ前に，リスト 14.3 の処理を行うなどしておくとよいでしょう．

リスト 14.3 ファントムイベントを除去する処理の例

```
while (spu_readchcnt(SPU_RdEventStat) > 0) {
    // 予想外のイベントの発生があった場合
    // 発生中のイベントを読み出してアクノリッジ
    unsigned d = spu_readch(SPU_RdEventStat);
    spu_writech(SPU_WrEventAck , d);
    // もしかしたら本当にイベントが発生しているかもしれないので
    // チャネルカウンタを参照して中身があれば読み捨てておく
    if (spu_readchcnt(SPU_RdSigNotify1) > 0)
        spu_readch(SPU_RdSigNotify1);
    if (spu_readchcnt(SPU_RdSigNotify2) > 0)
        spu_readch(SPU_RdSigNotify2);
}
```

あとがき

ここまで長々とお付き合いいただきありがとうございました．つい先ほど“ファンブックってレベルじゃねーぞw”と大変励みになる (？) 感想を査読陣の 1 人であるよみさんから頂いたわけですが，筆者の感じている SPE プログラミングの魅力，多少は伝わりましたでしょうか．

本書ができるまでを思い返してみると，本書のネタを最初に思いついたのは 2008 年の秋頃にまで遡ります．間に合えば PDF を CD に焼いて冬コミで誰かのブースの隅っこにでも置かせてもらえばいいかなーと 2008 年の冬頃に執筆を始め (当然書き上がらずそのときは手ぶらで一般参加)，その後しばらく中断した後，夏コミ当選の報せを受けて 2009 年 6 月頃から本格的に執筆を再開．そしてその年の 8 月の C76 で最初の版を頒布するに至りました．“こんな内容の本，コミケで買ってくれる人なんているんだろうか”と不安でいっぱいでしたが，蓋を開けてみれば刷った上下巻各 100 部はコミケ終了の 2 時間以上前に完売．その後もスペースに大勢の方が足を運んでくださり，コミックマーケット参加者の多様さをまざまざと見せつけられたのと同時に，“たくさん刷らなくてごめんなさい”と非常に申し訳ない気分になったものでした．

このとき買えなかった方のために，夏の本を増刷して通販することも一瞬考えたりもしたですが，夏の本の内容自体に色々と不満が残っていたこともあって，そうするよりもより一層内容を充実させた完成度の高いものを冬コミで出すべきだと考え，2009 年の 10 月頃から加筆・修正作業に取りかかりました．そして，その結果できあがったのが本書になります．今回も相当苦しみましたが，これを手に取って頂けているということは，無事どうにか本の形となって C77 会場で頒布できたということなのでしょう．

夏のバージョンでは上下巻で合わせて 152 ページだった本文は，100 ページ以上増えて 256 ページに膨れ上がりました．夏の本では時間の都合などで書けなかった内容を可能な限り盛り込み，特に SPE プログラミングの面白さをより肌で感じられるよう，実際の最適化・並列化の部分には相当力を入れました (と言う割には並列化の方は締め切り数日前に思いつくと同時に神が降臨して 1 日で書き上げたものでして，練りに練った内容というわけではなかったりもするのですが ⋯)．“ただメルセンヌ・ツイスタの乱数の合計値だけを求めればよい”という素晴らしい題材を与えてくださった Fixstars 社の Hack the Cell 2009 の中の人には，この場を借りて深く御礼申し上げます．まさに今回の本の目玉ともいうべき部分は，これがなかったらまったく書けなかったことでしょう．正直，並列化の話まで含めてこんなにうまくまとまるとは思いませんでした．

思えば夏コミの余韻冷めやらぬ 2009 年 8 月下旬，噂の新型 PS3 が発表され，“安く小さくなって，きっとこれでまた SPE プログラミングに興味を持つ人が増えるに違いない！”と盛り上

がった矢先，“新型の PS3 では Linux をインストールして動かすことができない” という衝撃の事実を知らされひどくショックを受けたものでした．せっかく安くスタイリッシュになったのに，Linux をインストールして趣味のプログラミングを楽しむといった用途で使えなくなってしまったのは非常に残念な話です．最初にこの報せを受けたときには完売の興奮が一気に吹き飛び，次回作への意欲が急激に低下したものでした．

その後，10 月には CELL REGZA が発表されたりもしましたが，11 月の終わり頃になって IBM が Cell 関連の開発から手を引くとか何とかいう噂が出たりして，“PS3 はそこそこ売れてるみたいだけど Cell/B.E. の未来は暗いなー” というのが正直なところです．

まあ，しかし，そうは言っても何かソフトが当たればこの先また盛り返すこともあるかもしれないですし，もしかしたら間違ってこの本を読んでしまった才能溢れる気鋭の開発者が，うっかりこれをきっかけにキラーアプリを作ってくれるかもしれないわけで，そう考えるとこんなときだからこそ応援すべきなんじゃないかとも思うわけです．

そしてそんなことを色々考えながら書いているうちに，ついつい気合が入ってしまい，気がつけばこんなに分厚い本ができあがってしまったのでした．この膨大な量の原稿のチェックを快諾して手伝ってくださった査読陣の皆様には足を向けて寝ることができません．本当にありがとうございました．

あ，次回作ですが，あるのかどうか何ともいえない感じです．もし何かあるとしても，さすがに SPE 関連の手持ちのネタはほとんど尽きたので，何か出すとしたら Cell/B.E. のプログラミング以外の本になるかと思います．たとえば SpursEngine の本とか …．まあでも SpursEngine は買ったはいいけどまだ一度も開発環境をいじったことがないので，いくら同じ SPE が搭載されているとはいえ，この状態で夏に同じくらいのクオリティの本が出せるのかというと相当厳しいでしょう．でも Linux 用のドライバも出たことだしなぁ．せっかくだから強力なキャプチャ環境も作って … いやでもそれには PC を新調しなければ … って心底どうでもいい話ですね．というわけで次回作がどうなるのか，あるのかどうかも含めて未定です．すいません．

そんなわけで，“ファンブック” とか称しながら表紙以外ガチガチのてふてふな教科書のような同人誌に最後までお付き合いいただき誠にありがとうございました．機会がありましたらまたどこかでお会いしましょう．それでは …

アリーヴェデルチ！ (さよならだ)

2009 年 12 月吉日
nosuke

索引

■ 記号・数字 ■

.timing ファイル 118
+∞ 方向への丸め 75, 77
--param 127
−0 74
-finline-limit 127
-funroll-loops 107
-lspe2 32
-m32 32
-O2 32
-O3 .... 31, 126
-S 118
-Winline 126
−∞ 方向への丸め 75, 77
/proc/cpuinfo 128
__attribute__((aligned(n))) 161
__attribute__((always_inline)) . 126
__attribute__((noinline)) 126
__builtin_expect 96
__lwarx 181
__send_to_ppe 228, 233
__stwcx 181
_divf4 78
0D 119
0d 119
1D 119
1d 119
2 命令同時発行 73, 119
32bit バイナリ 32
3 項演算子 97
64bit バイナリ 32

■ A ■

ai 101
argp 29, 33, 140

■ B ■

Barcelona SuperComputing Center → BSC
binfmt_misc 34
bisled 238
bisledd 238
bislede 238
brnz 97
brz 96
BSC 6

■ C ■

C99 .... 77, 133, 223
CBEA 1, 9
Cell Broadband Engine .... → Cell/B.E.
Cell Broadband Engine Architecture .... → CBEA
CELL REGZA 2
Cell SDK 5, 17, 24, 77, 239
Cell/B.E. 1, 9
ceqi 96
cgti 97

■ D ■

divf4 78
DMA 転送 11, 155, 188
—サイズ 155
—対象領域 155
—のアドレスの調整 161
—の完了を待つ 162
—の処理完了を確認 186
—の制約 156
—方向 156
—要求 156, 162, 165
—の完了状況取得 161
—を発行する (32bit 版) ... 158
—を発行する (64bit 版) ... 158
Dual Issue 73

■ E ■

EA 11, 14, 15
EA 空間から LS への DMA 転送を属性をつけて要求 185
EA 空間から LS への DMA 転送を要求 185
Effective Address .... → EA
EIB 10
Element Interconnect Bus . → EIB
elfspe2 34
envp 29, 33

■ F ■

Fedora 9 .... 5, 18, 20, 21
FIFO 144
First-Level Interrupt Handler .. → FLIH
Fixstars 80
FlexIO 10
FLIH 239
FPSCR 13, 54, 75, 76

■ G ■

GCC 5, 31–33, 161
GCC 4.1.1 33
GCC 4.3.2 33
genrand_mine 84, 191
GET 156
getllar 174–176, 178, 179, 237
getllar/putllc 174, 181

■ H ■

Hack the Cell 2009 ... 80, 83, 191
hbrr 99
Hello World 27

■ I ■

IEEE 754 .... 73, 75
IEEE 非準拠結果 75, 77
il 101
inline 126
intrinsic 35, 42

■ K ■

kboot: 21

■ L ■

L2 キャッシュ 10, 186
large-function-growth 127
Least Significant Bit ...... → LSB
libspe2 6, 27, 32, 34, 132, 153, 223
libspe2.h 28
Linux 5, 129
—カーネル 132
—内で SPE のコンテキストを生成する 130
LL/SC 174, 181
LL 命令 174
Load Linked/Store Conditional → LL/SC
Local Storage .... → LS
Local Store .... → LS
lqx 102
LS 11, 12
—から EA 空間への DMA 転送を属性をつけて要求 185
—から EA 空間への DMA 転送を要求 184
—の EA を取得する 182
LSB 38
lwarx 181

■ M ■

malloc.h 162
memalign 162
Memory Flow Controller . → MFC
Memory-Mapped I/O .. → MMIO
MFC 11, 155
—コマンド . 143, 156, 157, 168, 186
—タグ 143, 156
—タググループステータス更新 236
—のコマンドキュー空き発生 .. 236
—マルチソース同期 143
MFC_BARRIER_CMD 168
MFC_ClassID_CMD 211, 212
MFC_Cmd 157, 186, 236
MFC_CMDStatus 211, 212
MFC_COMMAND_QUEUE_AVAILABLE_EVENT 236
MFC_DECREMENTER_EVENT . 237
MFC_EAH 157, 211
MFC_EAL 157, 211
MFC_GET_CMD 158
MFC_GETB_CMD 168
MFC_GETF_CMD 168
MFC_GETL_CMD 171
MFC_GETLB_CMD 171
MFC_GETLF_CMD 171
MFC_GETLLAR_CMD 175
MFC_GETLLAR_STATUS ... 176
MFC_IN_MBOX_AVAILABLE_EVENT 237
mfc_list_element_t 169
MFC_LIST_STALL_NOTIFY_EVENT 237
MFC_LLR_LOST_EVENT ... 180, 237
MFC_LSA 157, 211
MFC_MSSync 211
MFC_OUT_INTR_MBOX_AVAILABLE_EVENT 237
MFC_OUT_MBOX_AVAILABLE_EVENT 237
MFC_PUT_CMD 158
MFC_PUTB_CMD 168
MFC_PUTF_CMD 168
MFC_PUTL_CMD 171
MFC_PUTLB_CMD 171
MFC_PUTLF_CMD 171
MFC_PUTLLC_CMD 175
MFC_PUTLLC_STATUS 176
MFC_PUTLLUC_CMD 178
MFC_PUTLLUC_STATUS ... 177
MFC_PUTQLLUC_CMD 178
MFC_QStatus 211, 213
MFC_RdAtomicStat 176, 177
MFC_RdListStallStat 237
MFC_RdTagMask 159, 160
MFC_RdTagStat ... 159–161, 177, 236, 237

MFC_SIGNAL_NOTIFY_1_EVENT 237
MFC_SIGNAL_NOTIFY_2_EVENT 237
MFC_Size 157
MFC_Size_Tag 211, 212
MFC_TAG_STATUS_UPDATE_EVENT ... 236
MFC_TAG_UPDATE_ALL ... 160, 162
MFC_TAG_UPDATE_ANY .. 160, 162
MFC_TAG_UPDATE_IMMEDIATE 160
MFC_TagID 157
MFC_WrListStallAck 172
MFC_WrTagMask 159
MFC_WrTagUpdate 159–161, 236
MMIO 133, 209
MMU 11, 155
Most Significant Bit ..... → MSB
MSB 37
mt 84, 191

■ N ■

NaN 74
newlib 6, 33, 223
notify 169, 171, 172, 237
num_rand 84

■ O ■

ori 101
OR モード 152–154

■ P ■

PlayStation3 .... → PS3
POSIX.1 .... 133, 223
posix_memalign 162
PowerPC 1, 9
PowerPC 970 10
PowerPC Processor Element ... → PPE
PowerPC Processor Unit . → PPU
PowerXCell 8i 2, 76
PPE 1, 9, 10, 27, 181
PPU 10
ppu-gcc 32, 33, 181
ppu-gcc43 33
ppu-gdb 134
ppu32-gcc 32
Prxy_QueryMask 211, 214
Prxy_QueryType 211, 214
Prxy_TagStatus 214
PS3 .... 2, 5, 16
PS3 Linux 17, 129
PS3 Linux Distributor's Starter Kit 19, 20
pthread 135, 138
pthread_create 139
pthread_join 140
pthread_t 138
PUT 156
putllc 174–176, 178
putlluc 177
putqlluc 177
P 操作 178

■ Q ■

qNaN 75, 77
qw 65
qwbyte 65
qwbytebc 65

■ R ■

RclassID 186
rid 186
RISC 174
Roadrunner 2

■ S ■

SC 命令 174
Second-Level Interrupt Handler → SLIH
selb 101
shli 102
si_sync 234
SIMD 10, 12
—演算 35
—化 77, 82
—数学ライブラリ 77
Single Instruction Multiple Data . → SIMD, → SIMD
SLIH 239
—を登録する 240
SMM 11, 14
SPE 1, 10, 14, 16, 27
—のコンテキストを実行する ... 29
—のコンテキストを作る 29
—のコンテキストを破棄する ... 30
—の実行ファイルを閉じる 30
—の実行ファイルを開く 28
—のプログラムを LS にロードする 29
spe_callback_handler_deregister ... 227
spe_callback_handler_query ... 228
spe_callback_handler_register . 227
SPE_CALLBACK_NEW 227
SPE_CALLBACK_UPDATE . 227
SPE_CFG_SIGNOTIFY1_OR . 154
SPE_CFG_SIGNOTIFY2_OR . 154
spe_context_create .. 29, 129, 133, 139, 151, 153
spe_context_destroy .. 30, 133, 140
spe_context_ptr_t 29, 138
spe_context_run 29, 133, 140
SPE_CONTROL_AREA 210
SPE_DEFAULT_ENTRY 29
SPE_EVENT_ALL_EVENTS . 150
spe_event_data_t 149
spe_event_handler_create 149
spe_event_handler_deregister .. 150
spe_event_handler_destroy 149
spe_event_handler_register 149
SPE_EVENT_IN_MBOX 150
SPE_EVENT_OUT_INTR_MBOX 150
SPE_EVENT_SPE_STOPPED 150
SPE_EVENT_TAG_GROUP . 150, 215
spe_event_unit_t 151
spe_event_wait 151
SPE_EVENTS_ENABLE 151, 228
spe_image_close 30, 140
spe_image_open 28, 139
spe_in_mbox_status 146
spe_in_mbox_write 145
spe_ls_area_get 182
SPE_MAP_PS .. 29, 133, 209, 215
SPE_MBOX_ALL_BLOCKING ... 145, 146
SPE_MBOX_ANY_BLOCKING ... 145, 146
SPE_MBOX_ANY_NONBLOCKING 145, 146
SPE_MFC_COMMAND_AREA ... 210
spe_mfc_command_area_t 210
spe_mfcio_get 185
spe_mfcio_getb 185
spe_mfcio_getf 185
spe_mfcio_put 184
spe_mfcio_putb 185
spe_mfcio_putf 185
spe_mfcio_tag_status_read 186
SPE_MSSYNC_AREA 210
spe_mssync_area_t 210
SPE_NO_CALLBACKS 30
spe_out_intr_mbox_read 146
spe_out_intr_mbox_status 147
spe_out_mbox_read 146, 147
spe_out_mbox_status 146
spe_program_handle_t 28, 29
spe_program_load 29, 139
spe_ps_area_get 210
SPE_SIG_NOTIFY_1_AREA . 210, 216
spe_sig_notify_1_area_t ... 210, 211
SPE_SIG_NOTIFY_2_AREA . 210, 216
spe_sig_notify_2_area_t ... 210, 211
SPE_SIG_NOTIFY_REG_1 ... 154
SPE_SIG_NOTIFY_REG_2 ... 154
spe_signal_write 153
spe_spu_control_area_t 210
spe_stop_info_t 30
SPE_TAG_ALL 186
SPE_TAG_ANY 186
SPE_TAG_IMMEDIATE 186
speid 140
spethread_t 138, 139
SPU 11, 12, 14
spu-gcc 31–33, 118
spu-gcc43 .... 33, 100, 173
spu-objdump 81
spu_absd 55
spu_add 47
spu_addx 47
spu_and 63
spu_andc 64
spu_avg 55
spu_bisled 238
spu_bisled_d 238
spu_bisled_e 238
spu_cmpabseq 56
spu_cmpabsgt 57
spu_cmpeq 56
spu_cmpgt 56
spu_cntb 57
spu_cntlz 58
spu_convtf 70
spu_convts 71
spu_convtu 71
spu_create 130
spu_eqv 63
spu_extend 71
spu_extract 45
spu_gather 58
spu_genb 49
spu_genbx 49
spu_genc 47
spu_gencx 47
spu_idisable 239
spu_ienable 239, 241
SPU_In_Mbox 216
spu_insert 46
spu_intrinsics.h 36, 142
spu_madd 50
spu_maskb 61
spu_maskh 62
spu_maskw 62
SPU_Mbox_Stat 216
spu_mfcdma32 .... 158, 170
spu_mfcdma64 ...... 158, 162, 170
spu_mfcio.h 158
spu_mfcstat 161, 162, 177, 188
spu_mffpscr 76

spu_mhhadd 53
spu_msub 50
spu_mtfpscr 76
spu_mul 50
spu_mule 51
spu_mulh 52
spu_mulo 51
spu_mulsr 53
spu_nand 63
spu_nmadd 51
spu_nmsub 51
spu_nor 63
spu_or 63
spu_orc 64
spu_orx 64
SPU_Out_Mbox 215
spu_promote 46
SPU_RdEventMask 235
SPU_RdEventStat . 180, 235, 238, 239
SPU_RdInMbox 145, 237
SPU_RdSigNotify1 . 153, 154, 237
SPU_RdSigNotify2 . 153, 154, 237
spu_re 54, 78
spu_readch 142
spu_readchcnt 142, 148
spu_rl 66
spu_rlmask 68, 70
spu_rlmaska 70
spu_rlmaskqw 68
spu_rlmaskqwbyte 69
spu_rlmaskqwbytebc 69
spu_rlqw 66
spu_rlqwbyte 66
spu_rlqwbytebc 66
spu_roundtf 71
spu_rsqrte 54
spu_run 131
spu_sel 59, 90, 100
spu_shuffle 60, 88
SPU_Sig_Notify_1 216
SPU_Sig_Notify_2 216
spu_sl 67
spu_slih_register 239–241
spu_slqw 67
spu_slqwbyte 67
spu_slqwbytebc 67
spu_splats 46
spu_sub 48
spu_subx 49
spu_sumb 55
spu_timer.h 239
spu_timing 118
SPU_WrEventAck .. 181, 235, 236, 238, 239
SPU_WrEventMask . 180, 235, 236
spu_writech 142
SPU_WrOutIntrMbox ... 145, 237
SPU_WrOutMbox 145, 237
spu_xor 63
spufs 129, 132
SpursEngine 2
SRR0 239
stop 223
stop-and-signal .. 30, 34, 223, 233
stwcx 181
sync 234
Synergistic Memory Management . → SMM
Synergistic Processor Element . → SPE
Synergistic Processor Unit → SPU

■ T ■

TclassID 186
Tempering 84, 92, 191
tid 186
timebase 128
TLB 186
Top500 2
type 223

■ V ■

vec_char16 39
vec_double2 39
vec_float4 39
vec_int4 39
vec_llong2 39
vec_short8 39
vec_uchar16 39
vec_uint4 39
vec_ullong2 39
vec_ushort8 39
volatile 125
V 操作 178

■ X ■

XDR 10, 155
XIO 10

■ あ ■

アウトオブオーダ実行 12
アウトバウンドメールボックス 144
アウトバウンド割り込みメールボックス . 144
アセンブリ言語 80, 81
アトミック更新 173, 178, 237
アドレス変換 11
アフィニティ属性 130
アライン 12, 39, 87, 161
アンダーフロー 75, 77
アンロック処理 181
イベント 143, 235
イベントハンドラ 149, 240
—のイベントを待つ 151
—の監視対象からイベントを削除する 150
—の監視対象にイベントを登録する 149
—を生成する 149
—を破棄する 149
イベントマスク 235, 241
インオーダ実行 12
インバウンドメールボックス 144
インライン展開 125
ウィークコンシステンシ 15
エントリポイント 29
オーバーフロー 75, 77
オペランド 43, 82

■ か ■

開発環境 4, 17
外部関数 34, 223
—呼び出し 133, 223
—に対するコールバック関数の登録を解除 227
—に対するコールバック関数を取得 228
—に対するコールバック関数を登録 227
—を呼び出す 228
型固有組み込み関数 42, 181
型総称組み込み関数 43, 142, 238, 239
機械語 81
奇数パイプ 12, 72, 119
奇数番目の要素 41
逆アセンブル 80, 126
逆数の近似値 54
ギャザ 169
キャリー 47
キュー 157, 165
共有データ 192
共有変数 174, 178
偶数パイプ 12, 72, 119
偶数番目の要素 41
クォータ 186
組み込み関数 35, 42
クリティカルセクション 178
ゲーム OS 5, 24
桁上がり 47
桁借り 49
コールバック関数 226
コヒーレントなメモリモデル 16
個別組み込み関数 42
コンテキスト 27, 138

■ さ ■

最近接値への丸め 75, 77
サイクル数 119
最適化 79
シグナル 34, 143, 152, 223
—1 .... 153, 237
—未処理データ発生 237
—2 .... 153, 237
—未処理データ発生 237
—のモード 153
—を書く 153
システムコール 130
実アドレス 11, 15, 209
実効アドレス 15
受信メールボックス 144–146, 237
—に書き込む 145
—の FIFO の空きを得る 146
—未処理データ発生 237
除算 54
スーパースカラ 12
スカラ 38
—からビットマスク生成 (16bit → 128bit) 61
—からビットマスク生成 (4bit → 128bit) 62
—からビットマスク生成 (8bit → 128bit) 62
—変数 39, 72
—をそのままベクタに変換 46
—を複製してベクタに変換 46
スキャッタ 169
スケジューリング 130
スタート 184
ストール 141, 148, 169, 171
スレッド 135, 138
スロット 41
セマフォ 178, 179
—変数 178, 179
ゼロ除算 54, 75, 77
ゼロ方向丸め 74, 75, 77
総称組み込み関数 43
送信メールボックス 144–146, 237
—空き発生 237
—から読み出す 146
—のメッセージ数を得る 146
ソースオペランド 42, 43
即値 43, 82, 233
ソフトウェアパイプライニング 114

■ た ■

帯域制限 186
タグ .... → MFC タグ, → MFC タグ
タググループ 156
タグマスク 159, 162
ダブルバッファリング化 189, 203
単精度浮動小数点数 74
置換クラス ID 186
チャネル 14, 141, 157
—から値を読み出す 142
—に値を書き込む 142
チャネルアクセス命令 11, 14, 141, 142

チャネルカウンタ 141, 148
—の値を読む 142
ディレクティブ 161
データ依存 86
デクリメンタ 127, 143, 237
—1 周 237
デスティネーションオペランド . 42, 43
デバッガ 134
転送クラス ID 186
同期式イベント検出 237
動的分岐予測 95
トラップ 75

■ な ■
ニーモニック 82

■ は ■
倍精度浮動小数点数 75
排他制御 174, 179
ハイパーバイザ 16, 186
—モード 10
パイプライン 12, 13, 72, 118
端数 104
端数処理 94, 104
パック 38
バリア 167
—同期 189, 196, 198
汎用レジスタ 13
非正規化数 74, 77
非対称型マルチコアプロセッサ 10
ビッグエンディアン 10, 12, 36
ビットの番号 37
非同期式イベント検出 237
ファントムイベント 241
ブートローダ 20, 21
フェンス 167
複合組み込み関数 44, 158, 161
物理アドレス 11, 209
浮動小数点数 73
プリファードスロット 41
プリフェッチ 13
プロキシコマンド 211
プロキシコマンドキュー 213
ブロッキング型の関数 30, 131
ブロッキング型のチャネル 141
プロファイル 100
—フィードバック 100
プロブレムステート 209
—領域 ... 14, 29, 133, 187, 209
—のレジスタのグループごとの EA を取得 210
分岐潰し 100
分岐ヒント 95, 201
—命令 13, 96, 98, 234
分岐ペナルティ 13
分岐ミス 95
分岐命令 73
並列化 191
ベクタ 13, 38
—全体をバイト単位で左シフト . 67
—全体をバイト単位で左ローテート 66
—全体をバイト単位で論理右シフト 69
—全体をビット単位で左シフト . 67
—全体をビット単位で左ローテート 66
—全体をビット単位で論理右シフト 68
—同士のビット単位の AND・NAND 63
—同士のビット単位の OR・NOR 63
—同士のビット単位の XOR・XNOR 63
—の 4 つのワードスロットの OR . 64
—の 4 要素単位の和 55
—の一部要素のビット幅を倍に拡張 71
—の一方をビット反転 → ビット単位で AND 64
—の一方をビット反転 → ビット単位で OR 64
—の各要素同士の絶対値の大小比較 57
—の各要素の 1 が立っているビットの数の取得 57
—の各要素の LSB の抽出 58
—の各要素の MSB からの連続した 0 の数の取得 58
—の各要素の逆数 54
—の各要素の逆数の絶対値の平方根 54
—の各要素の絶対値が同一かどうか比較 56
—の各要素を算術右シフト 70
—の各要素を整数から単精度の浮動小数点数にキャスト 70
—の各要素を単精度の浮動小数点数から符号つきの整数にキャスト . 71
—の各要素を単精度の浮動小数点数から符号なしの整数にキャスト . 71
—の各要素を左シフト 67
—の各要素を左ローテート 66
—の各要素を論理右シフト 68
—の型 38
—の要素同士の値が同一かどうか比較 56
—の要素同士の掛け算 50
—の要素同士の掛け算 (奇数番目 × 偶数番目) → 左シフト 52
—の要素同士の掛け算 (奇数番目同士) 51
—の要素同士の掛け算 (奇数番目同士) → 算術右シフト 53
—の要素同士の掛け算 (偶数番目同士) 51
—の要素同士の掛け算 (偶数番目同士) → 足し算 53
—の要素同士の掛け算 → 足し算・引き算 50
—の要素同士の掛け算 → 足し算 → 符号反転 51
—の要素同士の掛け算 → 引き算 → 符号反転 51
—の要素同士の差の絶対値 55
—の要素同士の大小比較 56
—の要素同士の足し算 47
—の要素同士の足し算 (桁上がりつき) 47
—の要素同士の引き算 48
—の要素同士の引き算 (桁借りつき) 49
—の要素同士の平均 55
—の要素の置き換え 46
—の要素の抽出 45
—の要素を出力する 45
—の要素を倍精度から単精度に丸める 71
—リテラル 40
—をパターンを用いてバイト単位で混合 60
—をパターンを用いてビット単位で混合 59
—変数 39
ペンディングイベントレジスタ 235, 236
ボロー 49

■ ま ■
マイナスゼロ 74
待ち合わせ 140, 144, 147
マルチコアプロセッサ 1
丸め 74, 75
ミューテックス 179
無限大 74
命令バッファ 13
メインループ 104
メールボックス 143, 144
メッセージング機構 141
メモリ共有型 173
メモリコンシステンシ 15
メモリ保護 11, 155
メルセンヌ・ツイスタ 80, 83

■ ら ■
ライトバック 10
ラインサイズ 10, 163, 175
リザベーション .. 174, 177, 181, 237
リザベーション消失 237
リスト 169
リスト DMA 169, 170, 237
リスト DMA ストール発生 237
リテラル 40
リネーミング 111
リングバッファ 204
ループアンローリング 106
例外 75
ロード・ストア命令 73
ロック処理 179

■ わ ■
割り込み 235, 238
割り込み送信メールボックス . 144–146, 149, 237
—空き発生 237
—から読み出す 146
—のメッセージ数を得る 147

参考文献

[1] “Cell Broadband Engine^{TM} アーキテクチャ (Version 1.01)”
http://cell.scei.co.jp/j_download.html

[2] “Cell Broadband Engine^{TM} アーキテクチャ用 C/C++ 言語拡張 (Version 2.3)”
http://cell.scei.co.jp/j_download.html

[3] “Synergistic Processor Unit 命令セット・アーキテクチャ (Version 1.2)”
http://cell.scei.co.jp/j_download.html

[4] “SPE Runtime Management Library Version 2.2”
https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/1DFEF31B3211112587257242007883F3

[5] “Cell Broadband Engine Programming Handbook Including the PowerXCell 8i Processor Version 1.12”
https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/7A77CCDF14FE70D5852575CA0074E8ED

[6] “Cell Broadband Engine^{TM} アーキテクチャ用 SIMD 数学ライブラリ仕様書 Version 1.0”
http://cell.scei.co.jp/j_download.html

[7] “Cell Broadband Engine Registers Version 1.5”
http://cell.scei.co.jp/j_download.html

[8] 黒澤 泰彦, 渡辺 幸男, 田胡 治之 “次世代プロセッサ Cell Broadband Engine”, 東芝レビュー 61 巻 6 号 (2006 年 6 月号), pp.9–15
http://www.toshiba.co.jp/tech/review/2006/06/index_j.htm

[9] 増渕美生 “Cell プロセッサ”, STARC シンポジウム 2005 招待講演 (I) 講演資料
http://www.starc.jp/download/sympo2005-j.html

[10] David A.Patterson, John L.Hennessy “コンピュータの構成と設計 第 3 版”, 日経 BP 社, 2006

[11] 天野 英晴 “並列コンピュータ”, 昭晃堂, 1996

[12] 中森 章 “マイクロプロセッサ・アーキテクチャ入門”, CQ 出版社, 2004

[13] Samuel P. Harbison III, Guy L. Steele Jr. “C リファレンスマニュアル 第 5 版”, エスアイビー・アクセス, 2008

■ 中の人

nosuke (のすけ)

普通の魔法使い (2 年目).
某電機メーカー半導体系事業部でソフトウェア開発的業務に従事.
ゲームが好きなはずなのだが，このところ古いゲーム機・ゲームソフトの
収集・改造・修理ばかりでゲームそのもので全然遊べていない.
最近はヤフオクを中心にアーケードゲームの基板収集に精を出す日々.

著者の微妙な戦歴 (Cell/B.E. 関連)：

・ 第 1 回 Cell Speed Challenge　4 位
・ Hack the Cell '09　5 位

著者の微妙な創作物 (Cell/B.E. 関連)：

・ celltripper (2ch のトリップ検索プログラム)
・ SPE を 256 倍使うための本 力の 16 倍編 ・ 技の 16 倍編 (C76 頒布)

■ 絵師

しら「」

本当は RSX の方が好き．たぶんメガネだ.

■ 濃く手伝ってくれた人々

ysfk，よみ，LimeFlavor

(敬称略・適当順)

■ すぺしゃるさんくす

ヒタヒタ，もっこす，ぼすけ，まやいい，地獄牛，CiVi,
tacash，おぼの，gwa

(敬称略・適当順)

奥　付

SPE を 256 倍使い倒すための本
– 完結編 –

発行日：	2009 年 12 月 30 日　初版第 1 刷発行
	2012 年 8 月 11 日　初版第 5 刷発行
著者：	nosuke (のすけ)
	`sasugaanija@gmail.com`
発行：	電脳のツボ
	`http://garakuta.homelinux.org/`
印刷：	(株) ポプルス